Enabling JSON Document Stores in Relational Systems

Craig Chasseur
University Of Wisconsin
chasseur@cs.wisc.edu

Yinan Li
University Of Wisconsin
yinan@cs.wisc.edu

Jignesh M. Patel
University Of Wisconsin
jignesh@cs.wisc.edu

ABSTRACT

In recent years, "document store" NoSQL systems have exploded in popularity. A large part of this popularity has been driven by the adoption of the JSON data model in these NoSQL systems. JSON is a simple but expressive data model that is used in many Web 2.0 applications, and maps naturally to the native data types of many modern programming languages (e.g. Javascript). The advantages of these NoSQL document store systems (like MongoDB and CouchDB) are tempered by a lack of traditional RDBMS features, notably a sophisticated declarative query language, rich native query processing constructs (e.g. joins), and transaction management providing ACID safety guarantees. In this paper, we investigate whether the advantages of the JSON data model can be added to RDBMSs, gaining some of the traditional benefits of relational systems in the bargain. We present Argo, an automated mapping layer for storing and querying JSON data in a relational system, and NoBench, a benchmark suite that evaluates the performance of several classes of queries over JSON data in NoSQL and SQL databases. Our results point to directions of how one can marry the best of both worlds, namely combining the flexibility of JSON to support the popular document store model with the rich query processing and transactional properties that are offered by traditional relational DBMSs.

1. INTRODUCTION

Relational database systems are facing new competition from various NoSQL ("Not Only SQL") systems. While there are many varieties of NoSQL systems, the focus of this paper is on document store NoSQL systems such as MongoDB and CouchDB. These systems are appealing to Web 2.0 programmers since they generally support JSON (Javascript Object Notation) as their data model. This data model fits naturally into the type systems of programming languages like Javascript, Python, Perl, PHP, and Ruby. In other words, using JSON addresses the problem of object-relational impedence mismatch.

WebDB 2013 New York, New York USA

These NoSQL document stores also use the flexibility of JSON to allow the users to work with data without having to define a schema upfront. This "no-schema" nature eliminates much of the hassle of schema design, enables easy evolution of data formats, and facilitates quick integration of data from different sources. In other words, these systems are query-first and schema-later or schema-never systems.

As a result, many popular Web applications (e.g. Craigslist, Foursquare, and bit.ly), content management systems (e.g. LexisNexis and Forbes), big media (e.g. BBC), and scientific applications (e.g. at the Large Hadron Collider) today are powered by document store NoSQL systems like MongoDB [3] and CouchDB [7].

Furthermore, JSON is now a dominant standard for data exchange among web services. Today many web services APIs support JSON as a data interchange format, including the APIs for Twitter [39], Facebook [37], and many Google services [25]. Thus, a JSON-based document store has a strong appeal to programmers who want to communicate with these popular web services in their applications.

Despite their advantages, JSON document stores suffer some substantial drawbacks when compared to traditional relational DBMSs. The querying capability of these NoSQL systems is limited – they are often difficult to program for complex data processing, and there is no standardized query language for JSON. In addition, these leading JSON document stores don't offer ACID transaction semantics.

It is natural to ask whether there is a fundamental mismatch between using a relational data processing kernel and the JSON data model (bringing all the benefits described above). Many NoSQL advocates have argued that relational database management systems (RDBMSs) do not have a key role to play in these modern flexible data handling environments. The focus of this paper is to directly address that belief, by designing, developing, and evaluating a comprehensive end-to-end solution that uses an RDBMS core, but exposes the same JSON programming surface as the NoSQL JSON document stores. The central contribution of this paper is that with our approach, *a relational DBMS can be augmented with a layer that transparently exports the JSON model, thereby providing all the advantages of NoSQL JSON document stores, while providing higher performance and richer functionality (such as native joins, and ACID guarantees instead of BASE consistency).* We hope that our proof-of-concept efforts here will encourage a broader discussion in the database community about supporting JSON data with relational technology, and adding features traditionally associated with RDBMSs to JSON document stores.

The key contributions of this work are as follows: First, we carefully consider how the JSON data model can be supported on top of a traditional RDBMS. A key challenge here is to directly support the schema flexibility that is offered by the use of the JSON data model (see Section 2). We then develop an automated mapping layer called Argo that meets all of the requirements identified for JSON-on-RDBMS (see Section 3). Argo runs on top of a traditional RDBMS, and presents the JSON data model directly to the application/user, with the addition of a sophisticated, easy to use SQL-based query language for JSON, which we call Argo/SQL. Thus, programming against Argo preserves all the ease-of-use benefits that are associated with document store NoSQL systems, while gaining a highly usable query language and additional features that are naturally enabled by using an underlying relational system (including ACID transactions). Argo includes two alternative mapping schemes which we compare against each other.

Second, we present a micro-benchmark, called NoBench, to allow us to gain insights into various performance aspects of JSON document stores (see Section 4).

Finally, we implement Argo on two open-source RDBMSs (MySQL and PostgreSQL) and use NoBench to contrast the performance of Argo on these two RDBMSs with MongoDB – the leading NoSQL JSON document store system. We draw various insights from this detailed evaluation (see Section 5), and conclude that with Argo or a similar system, traditional RDBMSs can offer the best of both worlds. Namely, Argo on an RDBMS preserves the flexibility associated with using the JSON data model and often outperforms existing NoSQL systems, all while providing higher functionality (e.g. joins, ACID) than existing NoSQL systems. To the best of our knowledge, this is the first paper that has shown the feasibility and the advantage of adapting RDBMSs to meet the popular programming surface of JSON document stores.

2. BACKGROUND

2.1 The JSON Data Model

The JSON data model [14] consists of four primitive types, and two structured types. The four primitive types are:

- Unicode Strings, wrapped in quote characters.
- Numbers, which are double-precision IEEE floats.
- Boolean values, which are *True* or *False*.
- Null, to denote a null value.

The two structured types are:

- Objects, which are collections of attributes. Each attribute is a key (String) and value (any type) pair.
- Arrays, which are ordered lists of values. Values in an array are not required to have the same type.

A value in an object or array can be either a primitive type or a structured type. Thus, JSON allows arbitrary nesting of arrays and objects.

Figure 1 shows an example of two JSON objects. The first object has string attributes *name* and *rival*, a numeric *age* attribute, and a boolean *indicted* attribute. The object also has an attribute *kids*, which is an array consisting of three strings and an object (it is perfectly legal to mix value types in an array). The nested object in the *kids* array defines its own mapping of keys to values, and includes its own array of *kids* (recall that JSON can nest objects and arrays arbitrarily deep within each other).

```
{
    "name":     "George Bluth",
    "age":      58,
    "indicted": True,
    "kids": ["Gob", "Lindsay", "Buster",
        {
            "name": "Michael",
            "age":  38,
            "kids": ["George-Michael"]
        }],
    "rival":    "Stan Sitwell"
}

{
    "name":            "Stan Sitwell",
    "age":             "middle-aged",
    "charity_giving": 250120.5,
    "kids":            ["Sally"]
}
```

Figure 1: A pair of valid JSON objects.

2.2 Implication of using JSON in a DBMS

JSON-based NoSQL document stores, such as MongoDB [3] and CouchDB [7], store collections of JSON objects in lieu of relational tables, and use JSON as the primary data format to interact with applications. There are no restrictions on the format and content of the objects stored, other than those imposed by the JSON standard. This leads to several salient differences from the relational model, namely:

- **Flexibility and Ease-of-Use**: Since applications don't have to define a schema upfront (or ever), application writers can quickly start working with the data, and easily adapt to changes in data structure.
- **Sparseness**: An attribute with a particular key may appear in some objects in a collection, but not in others. This situation often arises in e-commerce data [5, 12].
- **Hierarchical Data**: Values may be arrays or objects, nested arbitrarily deep. In contrast to the relational approach of normalizing hierarchical data into separate tables with foreign-key references, hierarchical data is represented intensionally within the parent object [31].
- **Dynamic Typing**: Values for a particular key have no fixed data type, and may vary from object to object.

2.3 Current NoSQL JSON Document Stores

The current leading *native* JSON document stores are MongoDB and CouchDB. MongoDB is the current market leader in this category, and has a richer feature set than CouchDB. MongoDB has a query language, while CouchDB requires writing MapReduce-based "views". MongoDB also offers higher performance [2]: it is implemented in C++, has robust indexing support, uses a fast binary representation of JSON [1] internally, and uses a fast custom binary protocol to connect to client applications. In contrast, CouchDB is implemented in Erlang, does not support indices in the usual sense (although it supports materialized views), and offers only a text-based REST interface to clients, requiring frequent expensive JSON serialization and deserialization.

It should be noted that JSON data is sometimes stored in

flat key-value stores like Riak [9] and Cassandra [8]. However, such systems don't provide the same intuitive programming surface as true JSON document stores, and there is an impedance mismatch between the application data model and the underlying database data model making it harder to write the end application program. Consequently, a number of large web applications are powered by native JSON document stores [4, 18, 20].

2.4 JSON vs. XML

JSON is similar to XML in that both are hierarchical semi-structured data models. In fact, JSON is replacing XML in some applications due to its relative simplicity, compactness, and the ability to directly map JSON data to the native data types of popular programming languages (e.g. Javascript), rather than having to programatically interact with an XML document via an API like DOM or SAX.

When XML emerged as a popular standard for data exchange, the database community responded with research into presenting relational databases as XML [11, 19, 35], efficiently storing and querying XML data using an underlying relational database [6, 15, 23, 26, 27, 34, 36, 38], and creating native non-relational XML stores [29]. We drew on this rich body of research into XML as a data model (especially systems for mapping and querying XML in a relational database) when designing Argo, our solution for storing and querying JSON data in a relational database. We discuss this XML-oriented research further in Section 6.

Although JSON is similar to XML in some ways, there are several major differences in the data models which cause previously developed XML mapping techniques not to be directly applicable to JSON. These differences include:

- **Schema**: In some applications of XML, an XML schema or document type definition (DTD) is available which describes restrictions on the structure of XML documents. Some XML-on-relational mapping techniques, such as the normalized universal approach of [23], the generation of overflow mappings in STORED [15], and the schema annotations which enable inlining in ShreX [6], exploit information from a schema for a class of XML documents to store certain data more efficiently and to reduce the number of joins that are required to reconstruct the XML documents. There is no equivalent to XML schema for JSON, so mappings must necessarily be schema-oblivious.
- **Repetition**: In a JSON object, a key maps to only one value (which may be a structured type nesting other values). In XML, a particular node type (identified by its tag name) may appear any number of times beneath a parent node (unless restricted by an available XML schema).
- **Ordering**: The order of nodes in an XML document (both the global order and the order among siblings) is potentially semantically meaningful and should be preserved. In JSON, order is only significant among the items in an array.
- **Typed Data**: The only primitive datatype that basic XML supports is a string (although the W3C standard for XML Schema defines a richer type system, and allows type constraints for certain values in an XML document to be expressed [30]). In contrast, JSON data is typed (see Section 2.1), with data types unambiguously determined from the JSON syntax.

We note that there is a rich body of work on schema-oblivious XML mapping and our mapping approach (outlined in Section 3) is inspired by that work, which unfortunately can't be directly applied to JSON. One could imagine developing a JSON-to-XML mapping layer, and using an existing XML mapping technique. Such a mapping must preserve JSON type information and allow queries to access attributes of different types (for instance, a comparison predicate should compare string values according to their lexicographical order, and numeric values according to their sign and magnitude). We could devise a mapping scheme which represents JSON values of different types as different tags in an XML document. Such a scheme would preserve the type information from JSON, but in many cases type-aware querying would require constructing complicated XPath expressions with different cases for different types, which in the case of non-string values would also require frequent deserialization of values from their string form. For these reasons, we have chosen to focus directly on implementing a JSON-on-relational system rather than considering a multi-layer JSON-on-XML-on-relational approach, although we do take inspiration from the previous work dealing with XML data in relational systems and selectively adapt several techniques from the XML research to the JSON model.

Finally, we note that in practice, the way JSON is deployed and used differs from the typical use cases for XML. JSON objects are lightweight, often with only a few levels of nesting, and each object usually represents a single entity in the data domain. In contrast, XML documents are often large, with many entities belonging to the same conceptual group being represented as repeated tags in a single document. As noted above, JSON data is accessed and manipulated by client applications directly as native variables, while more heavyweight XML documents are accessed via an API like DOM or SAX, or transformed by a rules-based mechanism like XSLT. The practical differences in the way JSON is used as compared to other data models motivated us to develop our own synthetic benchmark, NoBench, which models common JSON data usage patterns (see Section 4).

3. ARGO: BRINGING JSON TO RELATIONS

Existing NoSQL document stores are limited by the lack of a sophisticated and easy-to-use query language. The most feature-rich JSON-oriented query language today is MongoDB's query language. It allows selection, projection, deletion, limited types of updates, and `COUNT` aggregates on a single collection of JSON objects, with optional ordering of results. However, there is no facility for queries across multiple collections (including joins), or for any aggregate function other than `COUNT`. Such advanced query constructs must be implemented outside of MongoDB, or within MongoDB by writing a Javascript MapReduce job (while in other systems like CouchDB, even simple queries require MapReduce). MongoDB's query language requires specifying projection lists and predicates as specially-formatted JSON objects, which can make query syntax cumbersome.

NoSQL systems typically offer BASE (basically available, soft state, eventually consistent) transaction semantics. The BASE model aims to allow a high degree of concurrency, but it is often difficult to program against a BASE model; for example, it is hard for applications to reconcile changes [33]. Recent versions of NoSQL systems such as MongoDB have made an effort to improve beyond BASE, but these improvements are limited to ensuring durability of individual writes

and still fall far short of full ACID semantics.

To address these limitations, we developed Argo, an automated mapping layer for storing and querying JSON data in a relational DBMS. Argo has two main components:

- A mapping layer to convert JSON objects to relational tuples and vice versa. (Described in Section 3.1)
- A SQL-like JSON query language, called Argo/SQL, for querying JSON data. Beneath the covers Argo/SQL converts queries to vanilla SQL that works with the mapping scheme, and reconstructs JSON objects from relational query results. (Described in Section 3.2)

Note that since the Argo approach uses a relational engine, it can provide stronger ACID semantics.

3.1 Argo: The Mapping Layer

The storage format of Argo handles schema flexibility, sparseness, hierarchical data, and dynamic typing as defined in Section 2.2. The Argo storage format was designed to be as simple as possible while still being a comprehensive solution for storage of JSON data. There are indefinitely many possible storage formats, and we do not claim that ours is optimal; indeed, experience with XML suggests that the best-performing format is application-dependent [6]. (We do show in Section 5 that even this simple format provides good performance. We also consider exploring other mapping schemes to be an interesting direction for future work.)

In order to address sparse data representation in a relational schema, Argo uses a vertical table format (inspired by [5]), with columns for a unique object id (a 64-bit `BIGINT`), a key string (`TEXT`), and a value. Rows are stored in vertical table(s) only when data actually exists for a key, which allows different objects to define values for different keys without any artificial restrictions on object schema, and without any storage overhead for "missing" values.

To deal with hierarchical data (objects and arrays), we use a key-flattening approach. The keys of a nested object are appended to the parent key, along with the special separator character ".". This technique has the effect of making the flattened key look like it is using the element-access operator, which is conceptually what it represents. Similarly, each value in an array is handled by appending the value's position in the array to the key, enclosed by square brackets. This scheme allows the storage of hierarchical data in a single, flat keyspace. To illustrate, the flattened key for George Bluth's grandson "George-Michael" in Figure 1 is *kids[3].kids[0]*. Key-flattening is similar to the Dewey-order approach for recording the order of XML nodes [38] and the simple absolute path technique for representing a node's position in an XML document [36].

With the structured data types (objects and arrays) handled by key-flattening, we now address storage for the primitive types: strings, numbers, and booleans. We evaluated two storage schemes which we call Argo/1 and Argo/3.

3.1.1 Argo/1

This mapping scheme uses a single-table vertical format and retains the type information inside the table. Each collection of JSON objects is stored as a single 5-column table, which includes the standard object-id and key string columns, as well as one value column for each of the three basic JSON types: string (`TEXT`), number (`DOUBLE PRECISION`), and boolean (`BOOLEAN` or `BIT`). The value columns allow `NULL` values to mark columns that don't contain actual data values. It would also be possible to store any number of JSON collections in just one table by adding a column for a unique collection ID to this schema, but doing so can complicate query processing and cause data from a single collection to become sparse on data pages, incurring significant overhead. We therefore create one table for each JSON collection.

argo_people_data

objid	keystr	valstr	valnum	valbool
1	name	George Bluth	NULL	NULL
1	age	NULL	58	NULL
1	indicted	NULL	NULL	true
1	kids[0]	Gob	NULL	NULL
1	kids[1]	Lindsay	NULL	NULL
1	kids[2]	Buster	NULL	NULL
1	kids[3].name	Michael	NULL	NULL
1	kids[3].age	NULL	38	NULL
1	kids[3].kids[0]	George-Michael	NULL	NULL
1	rival	Stan Sitwell	NULL	NULL
2	name	Stan Sitwell	NULL	NULL
2	age	middle-aged	NULL	NULL
2	charity_giving	NULL	250120.5	NULL
2	kids[0]	Sally	NULL	NULL

Figure 2: Decomposition of JSON Objects from Figure 1 into the Argo/1 Relational Format.

We illustrate Argo/1 in Figure 2. All values are distributed into the rightmost three columns of the table, according to their types. Other fields are all filled with `NULL`. Attributes with the key *age* occupy two columns because they have two types in the sample JSON data. Values in nested arrays and objects have keys that are flattened as described above.

Reconstructing JSON objects from the mapped tuples is simple in this scheme. Argo starts with an empty JSON object and fetches rows from the table, ordered by `objid`. Argo examines the `keystr` of each row, checking for the element-access operator "." and/or an array subscript in square brackets, creating intermediate nested objects or arrays as necessary on the path represented by the key string. The sole non-null value from `valstr`, `valnum`, or `valbool` is inserted into the appropriate place in the reconstructed object. Argo then moves on to the next row. If the `objid` is the same as the previous row, it repeats the process of inserting that row's value. If the `objid` has changed, Argo emits the reconstructed JSON object and starts over with a new, empty object. After processing the last row, Argo emits the final object. If there are no rows to reconstruct objects from, Argo simply returns without emitting any objects.

3.1.2 Argo/3

This mapping scheme uses three separate tables (one for each primitive type) to store a single JSON collection. Each table has the standard `objid` and `keystr` columns, as well as a value column whose type matches the type of data (`TEXT` for string, `DOUBLE PRECISION` for number, and `BOOLEAN` or `BIT` for boolean). This is similar to a previously-studied schema for storing XML documents in object-relational databases which uses seperate "node tables" for different types of XML nodes (element, attribute, and text) [36]. Similar to Argo/1, each JSON collection is represented by its own set of 3 tables (again, we choose not to use a global set of 3 tables with a column for a unique collection ID to avoid extra complexity and overhead for storage and query processing).

argo_people_str

objid	keystr	valstr
1	name	George Bluth
1	kids[0]	Gob
1	kids[1]	Lindsay
1	kids[2]	Buster
1	kids[3].name	Michael
1	kids[3].kids[0]	George-Michael
1	rival	Stan Sitwell
2	name	Stan Sitwell
2	age	middle-aged
2	kids[0]	Sally

argo_people_num

objid	keystr	valnum
1	age	58
1	kids[3].age	38
2	charity_giving	250120.5

argo_people_bool

objid	keystr	valbool
1	indicted	true

Figure 3: Decomposition of JSON Objects from Figure 1 into the Argo/3 Relational Format.

Figure 3 illustrates the Argo/3 relational representation for the sample JSON objects from Figure 1. All values in the two objects in the file are distributed across three tables, according to their types. In particular, the attributes with key *age* have different value types in the two objects, and therefore are separately stored in table *argo_people_str* and *argo_people_num*. Just as in Argo/1, nested values are handled with key-flattening.

The object reconstruction process for Argo/3 is similar to the process for Argo/1, except that rows are read from the 3 tables in a collection in parallel, ordered by `objid`. A JSON object is reconstructed from all values across the 3 tables with the same `objid`, then the object is emitted and the Argo runtime moves on to the next-lowest `objid` from the three tables.

3.2 Argo/SQL

Argo/SQL is a SQL-like query language for collections of JSON objects. It supports three types of statements: `INSERT`, `SELECT`, and `DELETE`.

An insert statement has the following form:

```
INSERT INTO collection_name OBJECT {...};
```

Argo lazily creates collections when they are first accessed, so there is no `CREATE COLLECTION` statement. The way that Argo handles `INSERT` statements is described in Section 3.2.1.

A `SELECT` statement can specify a projection list of attribute names or * for all attributes. It can optionally specify a predicate in a `WHERE` clause. It is also possible to specify a single `INNER JOIN`. To illustrate these features, we show a number of valid Argo/SQL `SELECT` statements, and the results they return, in Figure 4. We discuss the evaluation of predicates in Section 3.2.2, selection and projection in Section 3.2.3, and join processing in Section 3.2.4.

Finally, a `DELETE` statement removes objects from a collection with an optional `WHERE` predicate. The following is an example of a valid `DELETE` statement in Argo/SQL:

```
DELETE FROM people WHERE "Lindsay" = ANY kids;
```

The above query deletes the first object from the collection. Note that it has a predicate that matches the string "Lindsay" with any of the values in the array *kids*. Array-based predicates are discussed in more detail in Section 3.2.2.

3.2.1 Insertion

Query	Result
`SELECT age FROM people;`	`{"age": 58}` `{"age": "middle-aged"}`
`SELECT * FROM people WHERE charity_giving > 100000;`	`{ "name": "Stan Sitwell", "age": "middle-aged", "charity_giving": 250120.5, "kids": ["Sally"] }`
`SELECT left.name, right.kids FROM people AS left INNER JOIN people AS right ON (left.rival = right.name);`	`{ "left": { "name": "George Bluth" }, "right": { "kids": ["Sally"] } }`

Figure 4: Argo/SQL SELECT statement examples.

When Argo receives an `INSERT` command, it begins a transaction in the underlying RDBMS (optionally, the user can choose to manually begin and later commit a transaction encapsulating more than just the insertion of a single object). Unique object IDs (the objid column) are generated from a SQL `SEQUENCE`. Argo recursively walks the structure of the JSON object, keeping track of the complete key-path (implementing key-flattening as described above) and executing an `INSERT` statement on the type-appropriate underlying table (for Argo/3), or type-appropriate columns of the underlying single table (for Argo/1) for each string, number, or boolean value encountered. When Argo has finished walking the object, it commits the transaction if the insert command is not in an explicit transaction block, thus guaranteeing that `INSERT`s of JSON objects are atomic.

3.2.2 Predicate Evaluation

Evaluating the `WHERE` clause of a `SELECT` or `DELETE` query requires a general mechanism for evaluating predicates on JSON objects and finding the set of `objid`s which match.

Simple Comparisons: Suppose we wish to evaluate a simple predicate comparing an attribute to a literal value. Argo selects `objid` from the underlying table(s) where `keystr` matches the specified attribute (values nested arbitrarily deep in arrays or objects are allowed), and where the value matches the user-specified predicate. Argo uses the type of the literal value in such a predicate to determine which table (in Argo/3) or column (in Argo/1) to check against. The six basic comparisons (`=`, `!=`, `<`, `<=`, `>`, `>=`), as well as `LIKE` and `NOT LIKE` pattern-matching for strings, are supported.

For example, the predicate `charity_giving > 100000` is evaluated in Argo/1 as:

```
SELECT objid FROM argo_people_data WHERE keystr =
"charity_giving" AND valnum > 100000;
```

In Argo/3, the same predicate is evaluated as:

```
SELECT objid FROM argo_people_num WHERE keystr =
"charity_giving" AND valnum > 100000;
```

Predicates Involving Arrays: Argo/SQL supports predicates that may match any of several values in a JSON array, using the `ANY` keyword. When `ANY` precedes an attribute name in a predicate, Argo will match any `keystr` indicating a value in the array, instead of exactly matching the attribute name. For example, the predicate `"Lindsay" = ANY`

kids is evaluated in Argo/1 as:

```
SELECT objid FROM argo_people_data WHERE keystr
SIMILAR TO "kids\[[0123456789]+\]" AND valstr =
"Lindsay";
```

And in Argo/3:

```
SELECT objid FROM argo_people_str WHERE keystr
SIMILAR TO "kids\[[0123456789]+\]" AND valstr =
"Lindsay";
```

Conjunctions: In general, an AND conjunction of predicates can not be evaluated on a single row of the underlying relational table(s), since each row represents only a single value contained in an object. Fortunately, Argo is able to take the intersection of the matching objids for the sub-predicates of a conjunction to find the matching objids for that conjunction. For example, the predicate age >= 50 AND indicted = True is evaluated with the following SQL statement in Argo/1:

```
(SELECT objid FROM argo_people_data WHERE keystr
= "age" AND valnum >= 50) INTERSECT (SELECT objid
FROM argo_people_data WHERE keystr = "indicted"
AND valbool = true);
```

In Argo/3, we again take the intersection of objids matching each sub-predicate. Sub-queries which evaluate the individual sub-predicates in a conjunction are constructed on the type-appropriate tables. The predicate age >= 50 AND indicted = True is evaluated with the following SQL statement in Argo/3:

```
(SELECT objid FROM argo_people_num WHERE keystr
= "age" AND valnum >= 50) INTERSECT (SELECT objid
FROM argo_people_bool WHERE keystr = "indicted"
AND valbool = true);
```

Disjunctions: Just as a conjunction can be evaluated as the intersection of its child subpredicates, a disjunction can be evaluated as the UNION of the sets of objids matching its child subpredicates. When using Argo/1, and when all of the disjunction's children are simple comparisons or other disjunctions whose leaf-level descendants are all simple comparisons, an optimization is possible: Argo simply connects the predicate clauses for each of the individual simple comparisons with OR.

For example, the predicate age >= 50 OR indicted = true is evaluated in Argo/1 as:

```
SELECT objid FROM argo_people_data WHERE (keystr
= "age" AND valnum >= 50) OR (keystr = "indicted"
AND valbool = true);
```

Argo/3 does not admit this same optimization, because in Argo/3 the individual predicates must be evaluated against 3 different tables [1]. So, in Argo/3, the same predicate must be evaluated with a UNION:

```
(SELECT objid FROM argo_people_num WHERE keystr =
"age" AND valnum >= 50) UNION (SELECT objid FROM
argo_people_bool WHERE keystr = "indicted" AND
valbool = true);
```

A disjunction whose children are not simple comparisons can not be evaluated with the OR optimization in Argo/1. The predicate charity_giving >= 100000 OR (indicted = True AND age >= 50) is evaluated in Argo/1 as:

```
(SELECT objid FROM argo_people_data WHERE keystr
= "charity_giving" AND valnum >= 100000) UNION
((SELECT objid FROM argo_people_data WHERE keystr
= "indicted" AND valbool = true) INTERSECT (SELECT
objid FROM argo_people_data WHERE keystr = "age"
AND valnum >= 50));
```

And in Argo/3 as:

```
(SELECT objid FROM argo_people_num WHERE keystr
= "charity_giving" AND valnum >= 100000) UNION
((SELECT objid FROM argo_people_bool WHERE keystr
= "indicted" AND valbool = true) INTERSECT (SELECT
objid FROM argo_people_num WHERE keystr = "age"
AND valnum >= 50));
```

Negations: Any basic comparison can be negated by taking the opposite comparison. For negations of conjunctions or disjunctions, Argo applies De Morgan's laws to push negations down to the leaf comparisons of a predicate tree.

3.2.3 Selection

A SELECT query requires reconstruction of objects matching the optional predicate. If there is a WHERE predicate in a SELECT query, the matching object IDs (found via the methods described in Section 3.2.2) are inserted into a temporary table (created with the SQL TEMPORARY keyword, so it is private to a connection), and attribute values belonging to the matching objects are retrieved via a SQL query of the following form in Argo/1:

```
SELECT * FROM argo_people_data WHERE objid IN
(SELECT objid FROM argo_intermediate) ORDER BY
objid;
```

In Argo/3, 3 similar queries fetch attribute values from the 3 tables comprising the collection:

```
SELECT * FROM argo_people_str WHERE objid IN
(SELECT objid FROM argo_intermediate) ORDER BY
objid;

SELECT * FROM argo_people_num WHERE objid IN
(SELECT objid FROM argo_intermediate) ORDER BY
objid;

SELECT * FROM argo_people_bool WHERE objid IN
(SELECT objid FROM argo_intermediate) ORDER BY
objid;
```

Note that it is possible to obtain the same results via a natural join of the data table(s) and the intermediate result table instead of a subquery (in fact, the join method is used in MySQL for performance reasons).

Argo iterates through the rows in the results of the above queries and reconstructs matching JSON objects according to the methods described in Sections 3.1.1 and 3.1.2.

Projection: The user may wish to project only certain attributes from JSON objects matching a query, as illustrated in the first query in Figure 4. The object reconstruction algorithms detailed above are the same, with the addition of a predicate that matches only the specified attribute names. Because a given attribute may be an embedded array or object, we can not simply match the attribute name exactly, we must also find all nested child values if any exist.

[1] A more limited optimization, where sub-predicates which specify literals of the same type are connected with OR, and three queries (one for each type-specific table) are UNIONed together, is possible in Argo/3.

Argo accomplishes this task with LIKE predicates on the key string column. For example, if the projection list contains the attributes "name" and "kids", Argo will run the following query (using Argo/1):

```
SELECT * FROM argo_people_data WHERE (keystr =
"name" OR keystr LIKE "name.%" OR keystr LIKE
"name[%" OR keystr = "kids" OR keystr LIKE
"kids.%" OR keystr LIKE "kids[%") AND objid IN
(SELECT objid FROM argo_intermediate) ORDER BY
objid;
```

Selection With No Predicate: In Argo/SQL, as in standard SQL, it is possible to have a SELECT query with no WHERE predicate. In this case, there is no predicate evaluation step, and projection and object reconstruction proceeds as detailed above, with the subquery for matching objid omitted.

3.2.4 Join Processing

Argo supports SELECT queries with a single INNER JOIN as illustrated by the last query in Figure 4. In order to evaluate a join condition on attributes of JSON objects, Argo performs a JOIN query on the underlying Argo/1 or Argo/3 tables where the keystrs match the names of the attributes in the join condition, and the values satisfy the join condition. For example, to evaluate the JOIN query shown in Figure 4, Argo would run the following (using Argo/1):

```
SELECT argo_join_left.objid,
argo_join_right.objid FROM argo_people_data AS
argo_join_left, argo_people_data AS argo_join_-
right WHERE argo_join_left.keystr = "rival" AND
argo_join_right.keystr = "name" AND argo_join_-
left.valstr = argo_join_right.valstr;
```

This is not the end of the story, however, as many join conditions make sense for more than one of the JSON primitive data types. The equijoin shown above evaluates the join condition for strings. To evaluate the condition for numbers and boolean values, Argo runs two more queries identical to the above, except with valstr replaced with valnum or valbool, respectively, and takes the UNION of the three.

For Argo/3, the join evaluation process is very similar, except that the three queries in the UNION each access the type-appropriate table.

As with a simple SELECT query, Argo stores the results of the above in a TEMPORARY intermediate table. Argo then selects rows matching the left and right objid in the intermediate table from the underlying data table(s) (optionally projecting out only certain attributes as described in Section 3.2.3), and reconstructs the joined objects according to the methods described in Sections 3.1.1 and 3.1.2.

3.2.5 Deletion

Just as with SELECT, a DELETE query may take an optional predicate, in which case object IDs matching the predicate are inserted into a temporary table. All of the attribute data belonging to matching objects is then deleted via a SQL query of the following form in Argo/1:

```
DELETE FROM argo_people_data WHERE objid IN
(SELECT objid FROM argo_intermediate);
```

When using Argo/3, there are 3 similar DELETE statements, one for each table.

The predicate evaluation query and the DELETE query are encapsulated in a single transaction.

If there is no predicate in a DELETE query, then all the objects in a collection are deleted via a statement similar to the above, with the subquery for matching objid omitted.

4. NOBENCH: A JSON DOCUMENT STORE BENCHMARK

To evaluate the performance of query processing in JSON document stores, we created the NoBench benchmark suite. NoBench follows the spirit of other database micro-benchmarks (such as [13,16]) in that it aims to identify a simple and small set of queries that touch on common operations in the target settings. The NoBench queries are not meant to replicate or simulate a particular production workload, but instead focus on individual operations common to many applications. NoBench consists of a data generator and a set of 12 simple queries that are defined below.

4.1 Generated Data

The NoBench data generator generates a series of JSON objects with the following attributes:

- str1, str2: a pair of unique strings.
- num: a unique number in the range 0 – N, where N is the total number of objects generated.
- bool: a boolean value (50% are true, 50% are false).
- dyn1: a dynamically typed value that is equal to *num* in 95% of the objects, and equal to *str1* in the remaining 5%.
- dyn2: a dynamically typed value that is equal to either *str1*, *num*, or *bool* in an even 1/3 of objects.
- nested_arr: a nested array of 0 – 7 strings randomly sampled from the Brown Corpus of published American English text [24]. The Brown Corpus follows a Zipfian distribution, and is used to simulate "tag" or keyword search.
- nested_obj: a nested object with 2 attributes:
  - str: a unique string that is equal to *str1* from a different object in the generated set.
  - num: a unique number that is equal to *num* from a different object in the generated set.
- sparse_XXX: a series of sparsely-populated attributes from *sparse_000* to *sparse_999*. Each object has a series of 10 consecutively-numbered sparse attributes chosen from the set of 1000. Sparse attributes are chosen to form 100 "clusters", so that any two objects will share all of their sparse attributes or none of them. Specifically, any given object has sparse attributes *sparse_XX0* through *sparse_XX9*, where XX varies from object to object. The value for each such attribute is randomly chosen from a set of 10 strings.
- thousandth: an integer in the range 0 – 999, equal to *num* modulo 1000

The generated data set includes hierarchical data, dynamic typing, and sparse attributes.

4.2 Queries

The twelve NoBench queries are designed to test the performance of common operations in a JSON document store. The final result for each query is inserted into a temporary

table or collection. The queries are grouped into five categories, and are described below. The full Argo/SQL and MongoDB commands for each query are shown in Table 1.

In all the queries below where the query has a literal constant (e.g. the boundaries of a range predicate), this constant is generated uniformly randomly (without substitution) from within the range of the domain for that literal. This method allows testing NoBench queries with different parameterization, touching different parts of the data set, so that we can more accurately gauge overall performance.

4.2.1 Projection: Q1 to Q4

Q1: This query projects the two common attributes *str1* and *num* from all the objects in the dataset. This query simply tests the ability of the system to project a small subset of attributes that are always present in each object.
Q2: This query projects the two nested attributes *nested_obj.str* and *nested_obj.num* from the entire dataset. Comparing to Q1 exposes the difference in projecting nested attributes.
Q3: This query projects two sparse attributes from one of the 100 clusters. This query tests the performance when projecting attributes that are defined in only a small subset of objects.
Q4: This query projects two sparse attributes from two different clusters. This query retrieves the same number of attribute values as Q3, but fetches them from twice as many objects (i.e. the cardinality of the result of Q4 is twice that of Q3).

4.2.2 Selection: Q5 to Q9

Note: Q5 below is a rifle-shot selection of a single object, while Q6 – Q9 each select 0.1% of the objects in the collection. Q6 – Q9 each measure the cost of reconstructing many objects that match a particular predicate, while the type of predicate is different for each query.
Q5: This query selects a single object using an match predicate on *str1*. This query tests the ability of the system to fetch a single whole object by an exact identifier.
Q6: This query selects 0.1% of the objects in the collection via a numeric range predicate on *num*. This query tests the speed of predicate evaluation on numeric values and the reconstruction of many matching objects.
Q7: This query selects 0.1% of the objects in the collection via a numeric predicate that selects values of *dyn1* in a particular range. Comparing this query with Q6 exposes the difference in evaluating a predicate on an attribute that is statically-typed and an attribute that is dynamically-typed.
Q8: This query selects approximately 0.1% of the objects in the collection by matching a string in the embedded array *nested_arr*. This query tests the speed of evaluating a predicate that matches one of several values inside a nested array, and simulates a keyword-search operation.
Q9: This query selects 0.1% of the objects in a collection by matching the value stored in a sparse attribute. This query tests evaluation of a predicate on an attribute that only exists in a small subset of the objects.

4.2.3 Aggregation

Q10: This query selects 10% of the objects in the collection (by selecting a range of values for *num*), and `COUNT`s them, grouped by *thousandth*. The result will have 1000 groups, each of whose count is 0.01% of the total number of objects in the collection. This query tests the performance of the `COUNT` aggregate function with a group-by condition.

4.2.4 Join

Q11: This query selects 0.1% of objects in the collection (via a predicate on *num*) and performs a self-equijoin of the embedded attribute *embedded_obj.str* with *str1*. This query tests the performance of joins, and simulates following social-graph edges for a set of many users.

4.2.5 Insert

Q12: This query inserts objects, in bulk, amounting to 0.1% of total data size. This query tests the performance of inserting data, and measures the cost of object decomposition and index maintenance (where applicable).

5. EVALUATION

This section presents the results of NoBench on a leading NoSQL database MongoDB (version 2.0.0), and Argo (see Section 3) on two representative open-source database systems PostgreSQL (version 9.1.1) and MySQL (version 5.5.15).

We ran our experiments on a server with two 2.67GHz 6-core Intel Xeon X5650 processors, 24GB of DDR3 main memory, and a hardware RAID-0 consisting of four 7200RPM disks. The system runs Scientific Linux 5.3 (kernel 2.6.18).

In all the experiments, we simply run one query at a time. Thus, we present results with a concurrency level of one. In keeping with the design of the NoBench (a micro-benchmark), this method allows us to determine the performance of core data processing operations without other compounding factors. In future work (see Section 7), we intend to investigate performance in higher concurrency and clustered environments. We do note, however, that data sharding for clustered environments is not an intrinsic advantage of MongoDB, or of NoSQL systems in general. It has been shown that hash-based sharding is easily adaptable to traditional relational databases, and that the resulting sharded RDBMS (specifically Microsoft SQL Server) typically outperforms sharded MongoDB [22].

5.1 Experimental Setup

In this section, we describe how each system is set up in order to run the queries in NoBench and obtain meaningful results. Since many parameters of MongoDB are not tunable, we configured PostgreSQL and MySQL correspondingly to ensure that all three systems are fairly compared. Note that, while we tuned the checkpointing and logging behavior of the three systems to be as similar as possible, MySQL and PostgreSQL both provide full ACID semantics for transactions, whereas MongoDB is primarily a BASE system and only guarantees durability of individual write operations. The relational systems may experience a disadvantage compared to MongoDB due to the overhead of their more sophisticated transaction management capabilities.

Buffer Pool. MongoDB does not have a standard "buffer pool," and instead uses memory-mapped files to access data. The server was dedicated to MongoDB during testing and we found MongoDB used up to 97.8% of the system memory. In contrast, PostgreSQL and MySQL manage memory

Query	MongoDB Command	Argo/SQL Command
Q1	`db["nobench_main"].find({}, ["str1", "num"])`	`SELECT str1, num FROM nobench_main;`
Q2	`db["nobench_main"].find({}, ["nested_obj.str", "nested_obj.num"])`	`SELECT nested_obj.str1, nested_obj.num FROM nobench_main;`
Q3	`db["nobench_main"].find({ "$or" : [ { "sparse_XX0" : {"$exists" : True} } , { "sparse_XX9" : {"$exists" : True} } ] }, ["sparse_XX0", "sparse_XX9"])`	`SELECT sparse_XX0, sparse_XX9 FROM nobench_main;`
Q4	`db["nobench_main"].find({ "$or" : [ { "sparse_XX0" : {"$exists" : True} } , { "sparse_YY0" : {"$exists" : True} } ] }, ["sparse_XX0", "sparse_YY0"])`	`SELECT sparse_XX0, sparse_YY0 FROM nobench_main;`
Q5	`db["nobench_main"].find({ "str1" : XXXXX })`	`SELECT * FROM nobench_main WHERE str1 = XXXXX;`
Q6	`db["nobench_main"].find({ "$and": [{ "num" : { "$gte" : XXXXX } }, { "num" : { "$lt" : YYYYY } }]})`	`SELECT * FROM nobench_main WHERE num BETWEEN XXXXX AND YYYYY;`
Q7	`db["nobench_main"].find({ "$and": [{ "dyn1" : { "$gte" : XXXXX } }, { "dyn1" : { "$lt" : YYYYY } }]})`	`SELECT * FROM nobench_main WHERE dyn1 BETWEEN XXXXX AND YYYYY;`
Q8	`db["nobench_main"].find({ "nested_arr" : XXXXX })`	`SELECT * FROM nobench_main WHERE XXXXX = ANY nested_arr;`
Q9	`db["nobench_main"].find({ "sparse_XXX" : YYYYY })`	`SELECT * FROM nobench_main WHERE sparse_XXX = YYYYY;`
Q10	`db["nobench_main"].group({"thousandth" : True}, { "$and": [{ "num" : { "$gte" : XXXXX } }, { "num" : { "$lt" : YYYYY } }]}, { "total" : 0 }, "function(obj, prev) { prev.total += 1; }")`	`SELECT COUNT(*) FROM nobench_main WHERE num BETWEEN XXXXX AND YYYYY GROUP BY thousandth;`
Q11	*Implemented in JavaScript MapReduce.*	`SELECT * FROM nobench_main AS left INNER JOIN nobench_main AS right ON (left.nested_obj.str = right.str1) WHERE left.num BETWEEN XXXXX AND YYYYY;`
Q12	*Bulk-insert using* `mongoimport` *command-line tool.*	PostgreSQL: `COPY` *table* `FROM` *file*; MySQL: `LOAD DATA LOCAL INFILE` *file* `REPLACE INTO TABLE` *table*;

Table 1: NoBench query commands in MongoDB query language and Argo/SQL.

in their own pinned buffer pool. For both these RDBMSs, we set the size of the buffer pool to be 3/4 of the size of the system memory, with additional memory allocated for temporary storage. For PostgreSQL, we allocated 18GB to the buffer pool, 1GB to the temporary buffer, and 4GB to the maintenance working buffer. For MySQL, we allocated 18GB to the buffer pool, 4GB for heap (temporary) tables, and 1GB for the InnoDB additional memory pool.

Checkpointing. In MongoDB, checkpointing is implemented by invoking a fsync system call on memory mapped files. We configure MongoDB to periodically call fsync every 5 minutes. Similarly, we set the checkpoint interval to be 5 minutes for PostgreSQL and MySQL.

Logging. Journaling is available in MongoDB starting from version 1.8. Group commits are performed every 100ms for journaling in MongoDB. We also configured PostgreSQL and MySQL to perform group commits over 100ms.

Indices. Benchmark performance is sensitive to the number and types of indices. In all three systems, we created indices to enhance query performance where sensible. All three systems use B-tree indices.

In PostgreSQL and MySQL, we built indices on the `objid`, `keystr`, `valstr`, and `valnum` columns of the Argo/1 format (illustrated in Figure 2). For Argo/3 (illustrated in Figure 3), indices were built on the `objid` and `keystr` columns of each of the three tables, as well as the `valstr` column of the string table and the `valnum` column of the number table.

In MongoDB, indices are built on the values associated with a particular key. The MongoDB documentation recommends building indices to match queries that will actually be run, and we follow this recommendation. We build indices on *str1* (for Q5), *num* (for Q6 and Q10), *dyn1* (for Q7), and *nested_arr* (for Q8). We do not build indices on any sparse attributes (which might help Q9, and possibly Q3 and Q4), as that would require the creation of 1000 separate indices to support any possible parameterization of the queries.

5.2 Methodology

In each run of our benchmark, the dataset is first generated into raw text files. The database is loaded with the dataset, and indices are constructed on the data. Since index construction and loading is a one-time cost we do not show these costs for all three systems (although Q12 measures the cost of index maintenance when performing a bulk insert). Once loading and building indices is completed, we restart the database. We then run the queries and measure the response time. The results for each query are stored in a temporary table or collection.

For each query, 10 runs are performed. Each individual run of the queries Q3–Q11 in NoBench has its parameters randomized so that the results are not distorted by caches in the database system. Parameters are semirandom, but are repeatable across different systems and multiple experiments.

We only report results for "warm" queries, since this simulates the common operational setting. We discard the maximum and minimum values among the 10 runs for each query. We report the mean execution time of the remaining 8 runs. With few exceptions, the normalized coefficient of variation in execution times was less than 20%, and tended to be

smaller for longer-running queries. Complete data on variation in query execution times are presented in Table 4.

At each scale factor, and for each query, we use the mean execution time from MongoDB as a baseline. To compare the performance of Argo on MySQL and PostgreSQL with the MongoDB baseline, we calculate a unitless speedup factor, which is the mean execution time for MongoDB divided by the mean execution time for the other system. To summarize the overall results, we report the geometric mean [2] of the speedup factors for all 12 queries (the geometric mean is the appropriate, meaningful average for normalized unitless numbers [21]). It is important to note that, as a microbenchmark, NoBench does not replicate or simulate any "real" query workload, and instead contains queries which are meant to individually stress different operations in the database. The geometric mean of speedup factors does not, therefore, indicate that the combination of Argo and a particular database is definitively faster or slower than MongoDB. It should also be noted that, because the 12 NoBench queries are dominated by selection (five queries) and projection (four queries), the reported geometric mean will be more heavily influenced by these classes of queries than joins, aggregates, and inserts (which are represented by one query each). The geometric mean of speedup factors is a useful for making broad at-a-glance comparisons between systems, but, as we shall see, fully understanding the differences in performance between document stores requires looking at the results for the individual queries.

All experiments are run by a NoBench driver script running on the same server. The response times for each query that we report here are collected in the client process, which includes the cost of mapping layers for the relational approaches.

In order to study how the systems perform as the data size is scaled up, the queries in NoBench are evaluated over datasets whose cardinality is scaled by a factor of 4, from 1 million to 64 million objects. The total on-disk sizes of the datasets are shown in Figure 5. The 1-million and 4-million object datasets fit in memory for each database, while the 16 million object dataset exceeds the memory size for the relational systems (but not MongoDB), and the 64 million object dataset requires spilling to disk for all systems.

5.3 Implementation

Because of the lack of a standardized query language for JSON document stores, some effort was necessary to port NoBench to each of the specific systems that we tested. The full commands for each query are shown in Table 1.

MongoDB: For queries Q1 through Q10, MongoDB's native JSON-based query language was used. MongoDB does not natively support joins (in fact, MongoDB's documentation explicitly encourages denormalizing and duplicating data in hierarchical structures to avoid the need for joins). To implement Q11, we used MongoDB's integrated Javascript MapReduce system. Note that, in MongoDB, MapReduce jobs are unable to take advantage of indices, so the map step always requires a complete scan of a collection, which negatively impacts performance. For Q12, the `mongoimport` command-line tool was used to bulk-insert a file full of JSON objects.

[2] GM = $\sqrt[12]{\prod_{n=1}^{12} Q_n}$ where Q_n is the speedup factor for query N.

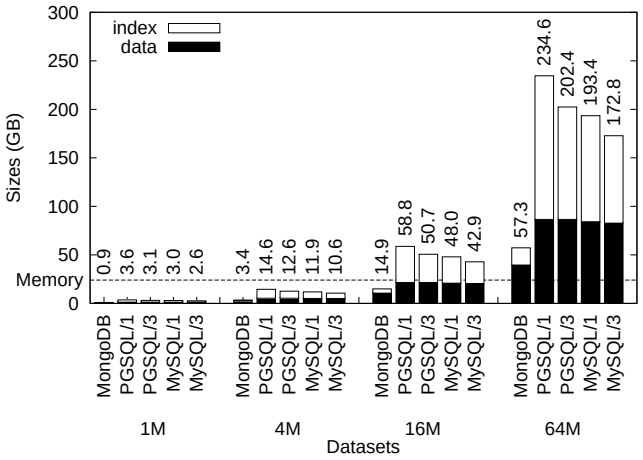

Figure 5: Database sizes. The numbers at the top of each bar indicates the full database size. The "/1" and "/3" indicates that the DBMS is using Argo/1 or Argo/3 respectively.

	PostgreSQL		MySQL	
	Argo/1	Argo/3	Argo/1	Argo/3
Q1	500.89	460.52 (**1.09**)	494.61	414.39 (**1.19**)
Q2	748.82	630.29 (**1.19**)	485.84	392.30 (**1.24**)
Q3	36.28	47.66 (**0.76**)	126.89	140.85 (**0.90**)
Q4	41.09	77.44 (**0.53**)	248.37	81.03 (**3.07**)
Q5	0.07	0.13 (**0.54**)	0.07	0.12 (**0.58**)
Q6	137.11	279.11 (**0.49**)	203.50	290.38 (**0.70**)
Q7	135.54	190.37 (**0.71**)	224.60	180.52 (**1.24**)
Q8	105.04	143.95 (**0.73**)	141.28	99.67 (**1.42**)
Q9	86.35	115.83 (**0.75**)	145.67	91.71 (**1.59**)
Q10	5163.81	35.57 (**145.17**)	2089.15	54.43 (**38.38**)
Q11	2245.96	2226.01 (**1.01**)	259.15	211.55 (**1.23**)
Q12	39.88	33.04 (**1.21**)	90.49	43.38 (**2.09**)
GM	**1.0**	**1.21**	**1.0**	**1.67**

Table 2: Argo/1 versus Argo/3 (16 million object dataset). All reported times are in seconds. Bold text in Argo/3 columns is speedup factor vs. Argo/1 on the same system (higher is better for Argo/3).

Relational DBMSs: When testing NoBench in relational systems (namely MySQL and PostgreSQL), we used the object decomposition/reconstruction and query evaluation strategies implemented in Argo.

5.4 Query Processing in Argo/1 and Argo/3

We have performed an extensive evaluation of Argo/1 and Argo/3 on both PostgresSQL and MySQL for all the datasets, and the complete results can be seen in Table 4. In the interest of space we only show one representative result here. Table 2 shows the results for the 16 million object dataset. As can be seen in this table, overall Argo/3 has higher performance than Argo/1 on both MySQL and PostgreSQL. To summarize our findings:

- The projection queries (Q1-Q4) tend to perform better with Argo/3 because the underlying RDBMS is able to

quickly see that there are no matching rows in the `bool` table, and the matching rows occupy fewer pages in the other two tables.
- The selection queries (Q5-Q9) are slower in PostgreSQL with Argo/3 as the object reconstruction step is more expensive, since it has to access data across multiple tables (though each table is more "compact", so has a more efficient index access). However, Argo/3 is sometimes faster than Argo/1 with MySQL as the database size is smaller with MySQL (see Figure 5), requiring fewer disk accesses and hence allowing MySQL to benefit from more compact indices.
- The aggregation query, Q10, is much faster with Argo/3, as the key component here is the access to the **num** values, and the smaller `num` table in Argo/3 (rather than a fat `data` table in Argo/1) results in a big performance improvement for Argo/3.
- For the join query (Q11), MySQL uses an index join algorithm that outperforms the sort-merge join algorithm that the PostgreSQL optimizer chose. Argo/3 performs somewhat better than Argo/1 on MySQL, because the indices used for the join are more compact, while Argo/1 and Argo/3 perform almost identically on PostgreSQL.
- Argo/3 outperforms Argo/1 on the bulk-insert Q12 since Argo/3 requires no storage space for `NULL`s, and fewer, smaller indices must be updated for each row.

We present a more detailed analysis of the performance of Argo/1 and Argo/3 in both MySQL and PostreSQL in Appendix A. Given the overall better performance with Argo/3, in the interest of space, for the remainder of this paper we only consider the Argo/3 mapping.

5.5 MongoDB vs. Argo

In this section, we present results from running NoBench on MySQL and PostgreSQL (with Argo/3), and MongoDB. The detailed results are shown in Table 3. At each scale factor, the geometric mean of speedup factors shows MySQL outperforming MongoDB. Argo on PostgreSQL outperforms MongoDB at the 1M and 4M scale factors, but falls somewhat behind at 16M and 64M. For all but the smallest scale, MySQL outperforms PostgreSQL.

Before we examine individual queries, we note that as a NoSQL system designed specifically for JSON data, MongoDB's architecture is substantially different from that of Argo running on MySQL or PostgreSQL. We found that a few major design differences between the systems had correspondingly major implications for performance:

- Storage format: MongoDB stores whole JSON objects contiguously using a fast binary representation of JSON [1]. Argo decomposes JSON objects into a 1-row-per-value format.
- Indices: indices in MongoDB are on the values of a particular attribute. Indices in Argo are built separately on `objid`, `keystr`, and various column values. MongoDB indices therefore tend to be smaller and are specialized to particular attributes.
- MapReduce: MongoDB can not natively query more than one table at once. To do even a simple self-join (as in Q11) requires writing a Javascript MapReduce job, which can not benefit from any indexing or native querying capability. Additionally, both map and reduce tasks in MongoDB operate on objects which are deserialized from MongoDB's native binary storage format and loaded into a Javascript VM, then have their output serialized and re-inserted into the database (i.e. each object makes two round-trips between the database and a dynamic language runtime).

5.5.1 Projection: Q1 – Q4

MongoDB's contiguous storage of objects actually hinders its performance compared to Argo when projecting attributes in Q1-Q4, since Argo can use an index to quickly fetch just the appropriate rows, while MongoDB must scan through every object and project two attributes out from each (the overhead of this projection operation is especially apparent for common attributes in Q1 and Q2, where it must be performed on every object). We have found that Argo/3 on MySQL performs competitively with MongoDB when projecting sparse attributes at the largest scale factors (16M and 64M objects), and substantially outperforms MongoDB for all other projection queries at all scales.

5.5.2 Selection: Q5 – Q9

For Q5 (rifle-shot selection), MongoDB scans to find the matching object, even though an index on *str1* is available to match the predicate. Argo on the SQL systems always uses an index to find the matching object, and hence Argo vastly outperforms MongoDB for the rifle-shot selection query at all scale factors. The fact that MongoDB does not take advantage of an obviously useful index in this query is a serious oversight by its query optimizer.

For the 0.1% selectivity queries (Q6-Q9) MongoDB does use indices. At a scale of 1M objects, Argo on the relational systems outperforms MongoDB. At 4M objects, MongoDB is more competitive, but Argo on MySQL still performs best. At 16M objects, MongoDB takes the lead over Argo on these queries. Because of MongoDB's compact contiguous storage format and relatively lightweight indices, MongoDB's complete dataset and indices fit in RAM at 16M objects, whereas Argo on MySQL and PostgreSQL must spill to disk (see Figure 5). At 16M objects, the object reconstruction stage of the selection queries in Argo must frequently access the disk, while MongoDB has no such handicap. At 64M objects, MongoDB maintains its advantage in these queries, as even though MongoDB's data is no longer totally memory-resident, it can still keep most of its indices in memory, and its contiguous storage format for objects allows it to fetch whole objects from the disk easily.

5.5.3 Aggregation: Q10

MongoDB's built-in `group` function was used to implement `COUNT` with `GROUP BY`, and we found that MongoDB generally tends to outperform Argo on MySQL (though not overwhelmingly) for Q10 up through a scale factor of 16M, while Argo on PostgreSQL has a small performance advantage at 1M and 4M objects. Once MongoDB's data no longer fits in memory, however, we found that MongoDB switched from a relatively fast, optimized in-memory implementation of the `group` function to an extremely slow MapReduce version. As such, MongoDB's performance for Q10 is much worse than Argo at 64M objects.

5.5.4 Join: Q11

As we saw above when MongoDB fell back on MapReduce for Q10 at 64M objects, MapReduce in MongoDB can

	1 Million Objects					4 Million Objects				
	Mongo	PGSQL	SU	MySQL	SU	Mongo	PGSQL	SU	MySQL	SU
Q1	104.56	2.01	**52.52**	5.05	**20.90**	433.87	9.95	**43.61**	22.32	**19.44**
Q2	102.95	2.30	**44.76**	4.97	**20.71**	419.47	10.11	**41.49**	20.65	**20.31**
Q3	3.24	0.03	**108.00**	0.05	**64.80**	13.00	0.31	**41.94**	0.27	**48.15**
Q4	3.53	0.06	**58.73**	0.05	**70.60**	15.55	0.31	**50.16**	0.26	**59.81**
Q5	1.59	0.001	**1590**	0.001	**1590**	6.20	0.06	**103.33**	0.01	**620**
Q6	1.76	0.06	**29.33**	0.17	**10.35**	8.32	44.02	**0.19**	3.88	**2.14**
Q7	1.58	0.05	**31.60**	0.11	**14.36**	6.27	23.15	**0.27**	2.46	**2.55**
Q8	1.66	0.05	**33.20**	0.11	**15.09**	6.72	16.00	**0.42**	1.91	**3.52**
Q9	1.90	0.17	**11.18**	0.47	**4.04**	8.02	11.78	**0.68**	3.06	**2.62**
Q10	1.24	0.85	**1.46**	2.28	**0.54**	4.81	3.90	**1.23**	12.12	**0.40**
Q11	211.81	16.69	**12.69**	0.16	**1323.8**	915.09	87.52	**10.46**	12.99	**70.45**
Q12	0.19	0.83	**0.23**	1.30	**0.15**	0.78	7.26	**0.11**	8.05	**0.10**
GM	**1.0**	–	**23.90**	–	**19.52**	**1.0**	–	**3.77**	–	**8.25**

	16 Million Objects					64 Million Objects				
	Mongo	PGSQL	SU	MySQL	SU	Mongo	PGSQL	SU	MySQL	SU
Q1	1706.39	460.52	**3.71**	414.39	**4.12**	7261.63	6004.00	**1.21**	1575.65	**4.61**
Q2	1728.25	630.29	**2.74**	392.30	**4.41**	7276.17	18913.28	**0.38**	1609.58	**4.52**
Q3	52.44	47.66	**1.10**	140.85	**0.37**	472.05	1811.96	**0.26**	514.90	**0.92**
Q4	62.77	77.44	**0.81**	81.03	**0.77**	553.92	1539.10	**0.36**	475.49	**1.16**
Q5	23.80	0.13	**183.08**	0.12	**198.33**	260.73	0.30	**869.1**	0.26	**1002.8**
Q6	31.86	279.11	**0.11**	290.38	**0.11**	353.10	2993.52	**0.12**	2438.96	**0.14**
Q7	23.91	190.37	**0.13**	180.52	**0.13**	321.47	2239.01	**0.14**	1912.36	**0.17**
Q8	25.32	143.95	**0.18**	99.67	**0.25**	326.38	2975.56	**0.11**	1048.50	**0.31**
Q9	31.24	115.83	**0.27**	91.71	**0.34**	362.65	1800.08	**0.20**	1019.05	**0.36**
Q10	18.26	35.57	**0.51**	54.43	**0.34**	31174.31	9479.01	**3.29**	2101.88	**14.83**
Q11	6651.15	2226.01	**2.99**	211.55	**31.44**	53318.21	10117.21	**5.27**	2543.00	**20.97**
Q12	8.13	33.04	**0.25**	43.38	**0.19**	35.12	123.31	**0.28**	22.82	**1.54**
GM	**1.0**	–	**0.93**	–	**1.08**	**1.0**	–	**0.80**	–	**2.35**

Table 3: Performance comparison across the three systems. All reported times are in seconds. The SU column indicates speedup relative to MongoDB at the same scale factor (higher is better). GM is the geometric mean of the speedup factor over all queries.

be terribly slow. The join query, Q11, also needs to use a MapReduce job, and as a result the performance of this join query is far worse than Argo.

5.5.5 *Bulk Insertion: Q12*

MongoDB outperforms Argo for bulk inserts at most scale factors. This is understandable, as MongoDB stores inserted objects contiguously (there is no need to decompose them) and the indices that need to be updated with each insert are comparatively lightweight. Nevertheless, Argo's insert times, especially at larger scale factors, are not so long compared to MongoDB to become a significant disadvantage.

5.6 Scaling

The results in Table 3 show that no system achieves linear scaleup across all scale factors. As might be expected, there is a significant performance penalty on each system when data no longer fits entirely in memory. Argo on the relational systems also suffers from more difficult object reconstruction as scale increases for Q6-Q9 and Q11 (as discussed in Sections 3.1.1 and 3.1.2), because it must access many (sometimes uncontiguous) rows when reconstructing matching objects.

5.7 Summation

Our evaluation shows that Argo/3 generally offers performance superior to Argo/1. When Argo/3 is run on MySQL, its performance is high enough to make it a very compelling alternative to MongoDB. We find that, when data is small enough to fit entirely in memory (as with 1M or 4M objects), Argo/3 on MySQL outperforms MongoDB across the board, except for `COUNT` and insertions, where Argo is nonetheless competitive. When data is too large to fit in memory, Argo/3 on MySQL has superior performance for projection, rifle-shot selection, aggregate, and join queries, but MongoDB performs better when selecting large groups of objects.

6. RELATED WORK

It has recently been shown that the object-graph data model of JSON (and of several other hierarchical data models used in NoSQL systems), wherein nested objects and values are represented intensionally inside of their parent objects, is the dual of the foreign-key (extensional) representation of parent-child relationships between entities in relational databases. This dual form of the relational model was christened CoSQL [31]. Proceeding from this realization, a generalized algebra for relational and co-relational data was developed, and a unified query interface for both was implemented in LINQ. One of the primary motivations for the work on CoSQL was the realization that the lack of a standardized data model and query language among NoSQL products, and hence the lack of interoperability between those products, is hindering the development and growth of the NoSQL market.

Currently, there is a nascent effort to define a standard-

ized declarative query language for JSON data, UnQL [28]. UnQL has the participation of developers from the SQLite and the CouchDB projects. The proposed UnQL syntax is inspired by SQL, and provides commands for insertion, deletion, selection, and updates over collections of JSON objects. Queries are limited to a single collection. Unlike Argo/SQL, the current draft specification does not support joins or aggregate functions.

It is hoped that the JSON data model, combined with the UnQL query language, will eventually be implemented across many NoSQL products, realizing the CoSQL authors' hopes for a compatible and competitive market for NoSQL databases. Our own work in developing Argo shows that a compatible implementation of UnQL based on a relational mapping layer is possible, and that such an implementation could be a compelling choice in such a market.

Argo/1 and Argo/3 were inspired by previous work which introduced a vertical representation for sparse data in relational systems [5]. Another approach to sparse data is an "interpreted" wide-row storage format, which has been implemented by modifying the storage layer of PostgreSQL [10].

The Dremel project introduced a columnar storage format for (fixed-schema) hierarchical data [32]. Each field in an object (including fields which may be repeated and/or nested arbitrarily deep) is mapped to a separate column-stripe, with integer repetition and definition levels attached to each value. It is important to note that the repetition and definition levels, which are necessary for object reconstruction, both depend on the data being statically typed so as to know where repeatable (i.e. array-like) and optional (i.e. sparse) fields exist. It follows that the Dremel approach is not directly applicable to dynamically typed data.

As we noted in Section 2.4, there is a large body of work on supporting XML in relational systems, which provided valuable insights in developing Argo.

Projects such as Silkroute [19] and Xperanto [11] provide sophisticated methods to expose queryable XML views of relational databases. Another approach extends SQL with new aggregate and scalar functions to emit XML documents as the result of SQL queries [35]. We note that functions for emitting JSON data from SQL queries are slated for inclusion in a forthcoming release of PostgreSQL [17], mirroring the latter approach (in a considerably simplified early form).

Other research has developed both schema-aware and schema-oblivious mappings for storing XML data in relational tables [6, 15, 23, 34], with careful planning to preserve the order of nodes in an XML document [38]. Building on these techniques, it was discovered that the use of relational index structures [26] and a new form of the join operator [27] substantially improves the performance of certain classes of XPath/XQuery queries over XML documents.

Compared to dealing with XML, JSON presents a unique set of challenges. Some aspects of JSON allow one to simplify techniques from the XML-mapping world, while some of the distinguishing aspects of JSON (listed in Section 2.4) make it different from XML when considering mapping schemes. Argo presents a simple and intuitive scheme for mapping JSON to a relational schema. Argo's simplicity makes it easy to implement and understand.

7. CONCLUSION AND FUTURE WORK

The database community has a long history of building robust and stable data products, but has been somewhat slow to respond to emerging application/data domains. The use of schema-less JSON document stores is rapidly gaining in popularity amongst developers of data-driven web and mobile applications, and traditional DBMSs are not being considered seriously in these new settings. Rather, developers are using emerging NoSQL JSON document stores.

In this paper, we have shown that with the mapping layer that we proposed, namely Argo, traditional RDBMSs can support the flexibility of the schema-less JSON data model. Furthermore, with Argo one can go beyond what JSON NoSQL systems offer today and provide an easy-to-use declarative query language. Our results demonstrate that the Argo solution is generally both higher performing and more functional (e.g. it provides ACID guarantees and natively supports joins) than the leading NoSQL document store MongoDB. With Argo, traditional RDBMSs can offer an attractive alternative to NoSQL document stores while bringing additional benefits such as administration and management tools that have been hardened for RDBMSs over time.

There are a number of directions for future work, including expanding the current evaluation to explore the impact of multi-user, multi-core, and cluster environments, and expanding the study to include distributed key-value stores and alternative relational mapping schemes.

8. REFERENCES

[1] 10Gen, Inc. BSON Specification. http://bsonspec.org/, 2011.

[2] 10Gen, Inc. Comparing MongoDB and CouchDB. http://www.mongodb.org/display/DOCS/Comparing+Mongo+DB+and+Couch+DB, 2011.

[3] 10Gen, Inc. MongoDB. http://www.mongodb.org, 2011.

[4] 10Gen, Inc. MongoDB Production Deployments. http://www.mongodb.org/display/DOCS/Production+Deployments, 2011.

[5] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *VLDB*, pages 149–158, 2001.

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

[7] Apache Software Foundation. Apache CouchDB. http://couchdb.apache.org/, 2011.

[8] Apache Software Foundation. The Apache Cassandra Project. http://cassandra.apache.org/, 2012.

[9] Basho Technologies, Inc. Riak. http://wiki.basho.com/Riak.html, 2012.

[10] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending rdbmss to support sparse datasets using an interpreted attribute storage format. In *ICDE*, pages 58–, 2006.

[11] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. Xperanto: Publishing object-relational data as xml. In *WebDB (Informal Proceedings)*, pages 105–110, 2000.

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

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[14] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[15] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with stored. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 431–442, New York, NY, USA, 1999. ACM.

[16] D. J. DeWitt. The wisconsin benchmark: Past, present, and future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[17] A. Dunstan. JSON for PostgreSQL 9.2. http://people.planetpostgresql.org/andrew/index.php?/archives/244-Under-the-wire.html, January 2012.

[18] E. Farrell. Erlang at the BBC. http://www.erlang-factory.com/conference/London2009/speakers/endafarrell, 2011.
[19] M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. Silkroute: A framework for publishing relational data in xml. *ACM Trans. Database Syst.*, 27(4):438–493, Dec. 2002.
[20] K. Finley. Why Large Hadron Collider Scientists are Using CouchDB. http://www.readwriteweb.com/enterprise/2010/08/lhc-couchdb.php, 2010.
[21] P. J. Fleming and J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, Mar. 1986.
[22] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the elephants handle the nosql onslaught? *PVLDB*, 5(12):1712–1723, 2012.
[23] D. Florescu and D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Rapport de recherche RR-3680, INRIA, 1999. Projet RODIN.
[24] W. Francis and H. Kucera. A standard corpus of present-day edited american english. *Department of Linguistics, Brown University*, 1979.
[25] Google, Inc. Using JSON in the Google Data Protocol. http://code.google.com/apis/gdata/docs/json.html, 2011.
[26] T. Grust. Accelerating xpath location steps. In *SIGMOD*, pages 109–120, 2002.
[27] T. Grust, M. van Keulen, and J. Teubner. Staircase join: teach a relational dbms to watch its (axis) steps. In *VLDB*, pages 524–535, 2003.
[28] R. Hipp, D. Katz, and B. Young. UnQL Specification. http://www.unqlspec.org/, 2011.
[29] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *VLDB*, 11(4):274–291, Dec. 2002.
[30] A. Malhotra and P. V. Biron. XML schema part 2: Datatypes second edition. W3C recommendation, W3C, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.
[31] E. Meijer and G. Bierman. A co-relational model of data for large shared data banks. *Queue*, 9:30:30–30:48, March 2011.
[32] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *PVLDB*, 3:330–339, September 2010.
[33] K. Muthukkaruppan. The Underlying Technology of Messages. Facebook's Engineering Notes, 2010.
[34] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of xml documents. In *The World Wide Web and Databases*, pages 137–150. Springer Berlin / Heidelberg, 2001.
[35] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as xml documents. In *VLDB*, pages 65–76, 2000.
[36] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and retrieval of xml documents using object-relational databases. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, DEXA '99, pages 206–217, London, UK, UK, 1999. Springer-Verlag.
[37] M. Slee. JSON and Other Fun Stuff. http://developers.facebook.com/blog/post/16/, 2007.
[38] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, pages 204–215, 2002.
[39] Twitter Inc. Twitter Developers Documentation. https://dev.twitter.com/docs, 2011.

APPENDIX

A. DETAILED PERFORMANCE ANALYSIS

In this section, we examine the performance differences between Argo/1 and Argo/3, and between PostgreSQL and MySQL, in greater detail. This analysis expands on the comparison between Argo/1 and Argo/3 in Section 5.4, and complements the comparison of Argo with MongoDB in Section 5.5.

In our evaluation, we used the `EXPLAIN` command provided by each database to determine the query plans which were being used to execute the underlying SQL generated by Argo, which in turn helped us to explain some of the differences in performance that we observed.

A.1 Projection

MySQL and PostgreSQL both use an index on the `keystr` column to find matching rows in either the Argo/1 or Argo/3 representation. Argo/3 tends to perform somewhat better for each of these queries, because the underlying relational systems are able to quickly see that there are no matching rows in the `bool` table, and the matching rows occupy fewer pages in the other two tables (because they are separated by type, and because there is no overhead for `NULL` values).

For the projection queries (Q1-Q4), PostgreSQL always uses a similar query plan: it uses an index on the `keystr` column to find matching rows in either the Argo/1 or Argo/3 representation. MySQL does the same for each of the queries which project sparse attributes (Q3, Q4), but behaves differently for Q1 and Q2. Using Argo/3, MySQL uses an index on the `keystr` column of the `str` and `bool` tables, but performs a simple scan on the `num` table, checking the `keystr` of every row. Using Argo/1, MySQL performs a simple scan on the `data` table and checks the `keystr` of every row. Because Q1 and Q2 project common attributes which occur in every object in the dataset, there are matching rows in almost every page of the `num` table of Argo/3 or the `data` table of Argo/1, and the selectivity of matching the `keystr` is very high (more than 10% in Argo/1, and more than 23% in the `num` table of Argo/3), so the MySQL query optimizer determines that using an index will not actually be helpful.

We can see that Argo/3 always outperforms Argo/1 when projecting common attributes. For MySQL, part of the difference is explained by the ability to use an index on `keystr` to speed up fetching results from the `str` table. Both MySQL and PostgreSQL need to access fewer pages, because they can each use an index on `keystr` to quickly determine that they do not need to retrieve any results from the `bool` table, and because all of the matching rows occupy fewer pages (because they are separated by type, and because there is no overhead for `NULL` values).

A.2 Selection

For the selection queries (Q5-Q9), PostgreSQL and MySQL use very similar query plans. Recall that a selection query generally involves 2 steps: a predicate evaluation step which finds `objid`s of matching objects, and an object reconstruction step which finds all rows with those `objid`s and reconstructs the original JSON objects from them. In each query Q5-Q9, MySQL and PostgreSQL both use the same strategy for the second (object reconstruction) step: they perform an indexed nested loops join between the intermediate result table and the single data table (for Argo/1) or each of the 3 data tables (for Argo/3).

The strategy used to evaluate the first (predicate evaluation) query varies from Q5-Q9, naturally, because the predicate varies. As with the object reconstruction step, MySQL and PostgreSQL use essentially the same query plan for each of the individual predicate evaluation queries. For Q5, the index on the `valstr` column (of the `data` table of Argo/1 or the `str` table of Argo/3) is used to match an exact value, and the single resulting row is filtered by `keystr` (this predicate

evaluation is always very fast, but Q5 takes slightly longer in total on Argo/3 because, during object reconstruction, rows belonging to the proper object must be fetched from at least 3 pages, from each of 3 tables, as opposed to 1 or 2 pages from a single table). For Q6 and Q7, an index on `valnum` is used to select numbers in the appropriate range, and the matching rows were filtered by `keystr`. For Q8, an index on `valstr` is used to select strings matching the keyword, and the matching rows are filtered by pattern-matching on `keystr`. For Q9, an index on `keystr` is used to match the appropriate attribute sparse attribute name, and an index on `valstr` is used to match the appropriate value, and the intersection of the rows matched by the two indices is taken.

A.3 Aggregation

Q10 shows the most extreme difference in performance between Argo/1 and Argo/3, with Argo/3 performing vastly better.

As a first step, both MySQL and PostgreSQL select 10% of the `objid`s in the dataset based on a range predicate on *num*. For Argo/1 and Argo/3, MySQL then uses an index on `keystr` to find rows representing *num* attributes, and then filter them based on valnum. In Argo/3, MySQL performs this operation on the `num` table, where about 23% of the rows hold *num* values (so there are comparatively more matches per page) and the table itself is far smaller than the single `data` table in Argo/1 (so there are fewer pages to visit). PostgreSQL uses an intersection of matches from the indices on `keystr` and `valnum` to find matching rows, but just like MySQL it will have to access many more pages in Argo/1 than in Argo/3.

For the second step, both MySQL and PostgreSQL join the intermediate result from the first step with either the single `data` table for Argo/1, or the `num` table for Argo/3, filtering the result to rows where `keystr` equals *thousandth* (the group-by attribute), and performing the aggregate on the result, grouping by `valnum` (i.e. the values of the attribute *thousandth*). Because the filtering occurs after the join, minimizing the size of the join result is crucial to performance. In Argo/3, the join is with the comparatively small `num` table, and over 23% of the resulting rows will match the filter and actually go on to be used in the aggregate. In Argo/1, the join is with the much larger `data` table (and this join is 100 times the size of the object reconstruction joins in Q6-Q9), and only about 5% of the resulting rows match the filter.

In both steps of Q10, there is a tremendous advantage to working with a smaller `num` table in Argo/3, rather than a fat `data` table in Argo/1.

A.4 Join

On Q11, both MySQL and PostgreSQL first use an index to select 0.1% of object ids. PostgreSQL uses an index on `keystr` to get rows corresponding to the join attributes *str1* and *nested_obj.str*, then performs a sort-merge join on `valstr`. MySQL, on the other hand, performs an indexed join on `valstr` first, then filters the result for `keystr`s matching the join attributes. Both databases then fetch rows belonging to the left and right objects in each result pair using the same sort of indexed nested loop joins described above for selection queries. There is not a great difference between Argo/1 and Argo/3 for joins, but MySQL does outperform PostgreSQL substantially, because the indexed join performed by MySQL turns out to be a great deal faster than the sort-merge join performed by PostgreSQL, even though MySQL must filter the join results afterwards.

A.5 Insertion

On Q12, the bulk-insert query, Argo/3 outperforms Argo/1. This is because, although the same number of rows are inserted for each format, Argo/3 requires no storage space for `NULL`s, and fewer, smaller indices must be updated for each row.

B. FULL EXPERIMENTAL RESULTS

In Table 4, comprehensive NoBench results for all queries at all scale factors are presented, including relative speedup between Argo/1 and Argo/3 on MySQL and PostgreSQL, and the normalized coefficient of variation for each experimental result. This table contains a superset of the information presented in Tables 2 and 3.

1 Million Objects										
	MongoDB		PostgreSQL				MySQL			
		CoV	Argo/1	CoV	Argo/3 (**SU**)	CoV	Argo/1	CoV	Argo/3 (**SU**)	CoV
Q1	104.56	0.001	2.12	0.040	2.01 (**1.05**)	0.074	12.80	0.003	5.04 (**2.54**)	0.177
Q2	102.95	0.001	2.20	0.091	2.30 (**0.96**)	0.175	12.77	0.002	4.97 (**2.57**)	0.080
Q3	3.24	0.064	0.03	0.012	0.03 (**1.00**)	0.401	0.14	0.195	0.05 (**2.80**)	0.004
Q4	3.53	0.079	0.04	0.128	0.06 (**0.67**)	0.201	0.12	0.152	0.05 (**2.40**)	0.006
Q5	1.59	0.087	0.17	0.097	0.001 (**170**)	0.827	0.001	0.108	0.001 (**1.00**)	0.024
Q6	1.76	0.064	1.54	1.547	0.06 (**25.7**)	0.019	0.18	0.176	0.17 (**1.06**)	0.111
Q7	1.57	0.079	0.04	0.031	0.05 (**0.80**)	1.569	0.15	0.140	0.11 (**1.36**)	0.026
Q8	1.66	0.054	0.04	0.004	0.05 (**0.80**)	0.186	0.14	0.156	0.11 (**1.27**)	0.190
Q9	1.90	0.086	0.15	0.027	0.17 (**0.88**)	0.081	0.48	0.202	0.47 (**1.02**)	0.140
Q10	1.24	0.068	1.54	0.094	0.85 (**1.81**)	0.125	5.54	0.082	2.28 (**2.43**)	0.081
Q11	211.81	0.009	14.97	0.010	16.69 (**0.90**)	0.157	0.17	0.147	0.16 (**1.06**)	0.038
Q12	0.19	0.345	0.90	0.104	0.83 (**1.08**)	0.398	1.62	0.246	1.30 (**1.25**)	0.319
GM			**1.0**		**1.94**		**1.0**		**1.59**	

4 Million Objects										
	MongoDB		PostgreSQL				MySQL			
		CoV	Argo/1	CoV	Argo/3 (**SU**)	CoV	Argo/1	CoV	Argo/3 (**SU**)	CoV
Q1	440.00	0.014	8.32	0.246	9.95 (**0.84**)	0.218	57.79	0.011	22.32 (**2.59**)	0.244
Q2	436.58	0.039	8.85	0.161	10.11 (**0.88**)	0.392	57.47	0.018	20.65 (**2.78**)	0.021
Q3	13.00	0.037	0.35	1.102	0.31 (**1.13**)	1.388	0.41	0.026	0.27 (**1.52**)	0.128
Q4	15.54	0.028	0.27	0.124	0.31 (**0.87**)	0.138	0.44	0.098	0.26 (**1.69**)	0.132
Q5	6.20	0.076	0.09	0.116	0.06 (**1.5**)	0.216	0.01	0.715	0.01 (**1.00**)	0.705
Q6	8.32	0.072	23.32	0.027	44.02 (**0.53**)	0.026	9.94	0.031	3.88 (**2.56**)	0.295
Q7	6.27	0.046	19.41	1.662	23.15 (**0.84**)	1.791	8.31	0.045	2.46 (**3.38**)	0.021
Q8	6.72	0.046	15.90	0.188	16.00 (**0.99**)	0.045	5.34	0.039	1.91 (**2.80**)	0.060
Q9	8.02	0.078	13.19	0.042	11.78 (**1.12**)	0.077	5.46	0.057	3.06 (**1.78**)	0.032
Q10	4.81	0.048	22.48	1.521	3.90 (**5.76**)	0.057	25.60	0.127	12.12 (**2.11**)	0.081
Q11	915.09	0.013	79.34	0.012	87.52 (**0.91**)	0.007	9.80	0.186	12.99 (**0.75**)	0.108
Q12	0.78	0.120	8.67	0.281	7.62 (**1.14**)	0.184	10.04	0.066	8.05 (**1.25**)	0.075
GM			**1.0**		**1.10**		**1.0**		**1.85**	

16 Million Objects										
	MongoDB		PostgreSQL				MySQL			
		CoV	Argo/1	CoV	Argo/3 (**SU**)	CoV	Argo/1	CoV	Argo/3 (**SU**)	CoV
Q1	1758.77	0.030	500.89	0.048	460.52 (**1.09**)	0.102	494.61	0.091	414.39 (**1.19**)	0.148
Q2	1726.37	0.001	748.82	0.028	630.29 (**1.19**)	0.080	485.84	0.004	392.30 (**1.24**)	0.007
Q3	52.44	0.012	36.28	0.403	47.66 (**0.76**)	0.142	126.89	0.201	140.85 (**0.90**)	0.057
Q4	62.77	0.021	41.09	1.215	77.44 (**0.53**)	0.444	248.37	0.211	81.03 (**3.07**)	0.597
Q5	23.80	0.013	0.07	0.331	0.13 (**0.54**)	0.180	0.07	0.192	0.12 (**0.58**)	0.081
Q6	31.86	0.053	137.11	0.403	279.11 (**0.49**)	0.294	203.50	0.036	290.38 (**0.70**)	0.281
Q7	23.91	0.022	135.54	0.107	190.37 (**0.71**)	1.041	224.60	0.028	180.52 (**1.24**)	0.061
Q8	25.32	0.013	105.04	0.130	143.95 (**0.73**)	0.419	141.28	0.041	99.67 (**1.42**)	0.073
Q9	31.24	0.019	86.35	0.069	115.83 (**0.75**)	0.087	145.67	0.087	91.71 (**1.59**)	0.046
Q10	18.26	0.009	5163.81	0.644	35.57 (**145.17**)	0.247	2089.15	0.963	54.43 (**38.38**)	0.099
Q11	6651.15	0.006	2245.96	0.689	2226.01 (**1.01**)	0.078	259.15	0.056	211.55 (**1.23**)	0.045
Q12	8.13	1.270	39.88	0.500	33.04 (**1.21**)	0.449	90.49	0.559	43.38 (**2.09**)	0.053
GM			**1.0**		**1.21**		**1.0**		**1.67**	

64 Million Objects										
	MongoDB		PostgreSQL				MySQL			
		CoV	Argo/1	CoV	Argo/3 (**SU**)	CoV	Argo/1	CoV	Argo/3 (**SU**)	CoV
Q1	7269.68	0.001	–	–	6004.00 (–)	–	625.17	0.214	1575.65 (**0.40**)	0.007
Q2	7298.89	0.003	–	–	18913.28 (–)	–	498.27	0.026	1609.58 (**0.31**)	0.005
Q3	472.95	0.001	–	–	1811.96 (–)	–	372.02	0.979	260.88 (**1.43**)	0.974
Q4	555.24	0.002	–	–	1539.10 (–)	–	1011.85	0.420	825.51 (**1.23**)	0.424
Q5	260.73	0.018	–	–	0.30 (–)	–	0.70	0.175	0.41 (**1.71**)	0.361
Q6	353.10	0.027	–	–	2993.52 (–)	–	1430.98	0.054	2728.33 (**0.52**)	0.106
Q7	321.47	0.037	–	–	2239.01 (–)	–	1429.88	0.020	2072.32 (**0.69**)	0.077
Q8	326.38	0.013	–	–	2975.56 (–)	–	658.37	0.004	1016.37 (**0.65**)	0.032
Q9	362.65	0.019	–	–	1800.08 (–)	–	639.70	0.059	1052.06 (**0.61**)	0.031
Q10	31174.31	0.000	–	–	9479.01 (–)	–	3344.70	0.000	2516.93 (**1.33**)	0.165
Q11	53318.21	0.000	–	–	10117.21 (–)	–	1588.37	0.092	2726.07 (**0.58**)	0.067
Q12	35.12	0.087	–	–	123.31 (–)	–	703.74	0.509	296.27 (**2.38**)	0.923
GM			**1.0**		–		**1.0**		**0.82**	

Table 4: Comprehensive results at all scale factors. All reported times are the mean of run times in seconds. Bold text in Argo/3 columns is speedup factor vs. Argo/1 on the same system (higher is better for Argo/3) (speedup vs. MongoDB is reported in Table 3). CoV columns are the normalized coefficient of variation of run times.