A JSON Document Algebra for Query Optimization

Tomas LLano-Rios[1], Mohamed Khalefa[2] and Antonio Badia[1]

[1]Computer Science and Engineering Department, University of Louisville, Louisville KY 40292 USA
[2]Old Westbury, State University of New York, NY, USA

Abstract
Query optimization in JSON database systems lacks the robustness of more mature databases. To address this, we propose an algebra to manipulate collections of JSON documents. The algebra is defined at 3 levels: the individual document level; the collection (core) level; and the collection (extended) level. The third level includes operators that can be defined in terms of the second level, but are useful for optimization. The operators in all levels are tailored to JSON data. We also define equivalences that allow the generation of multiple (logical) query trees for a single query, therefore providing the basis for cost-based query optimization for JSON query languages.

Keywords
JSON, query optimization, nested relational algebra

1. Introduction

The popularity of JSON as a data exchange format has led to the development of several data systems focused on JSON data (Asterix, MongoDB, CouchDB, and others). In spite of this, there is still relatively little work on query optimization for such systems. Several factors have contributed to this sparseness of results. First, there is no standard query language for JSON [1]; each system offers its own query language. This makes it hard to propose and analyze general solutions. Another factor is that JSON, as a data model, presents several issues that are difficult to deal with. The heterogeneity of the data, and its mixed structure presents challenges to developing a query language with formal semantics: JSON is different from other *hierarchical* models like nested relations [2] and XML [1], which means that past research on nested relational algebra (NRA) or query languages for XML cannot be used without a substantial revision [2].

The goal of this research is to design a query language that: a) is closed on JSON (takes only JSON data as input, produces only JSON data as output); b) is declarative and has formal semantics; and c) provides a good base for (logical) query optimization. Our approach is based on defining an algebraic language, i.e. one that provides operators that take as input (and produce as output) JSON data. We divide the definition of the algebra in 3 levels: level 1 manipulates individual documents; level 2 manipulates collections (defined as multisets of documents), and level 3 provides additional operators for collections that are not primitive (they can be defined using level 2 operators) but are helpful for optimization. Finally, we provide a set of equivalences for the algebra that can be used to generate alternative query plans for a given query. We stress that, even though the operators resemble those of Nested Relational Algebra, they are defined anew and tailored to JSON data. Thus, although some of the equivalences shown extend known relational equivalences to the JSON framework (others are new), they had to be proven from scratch.

The rest of this paper is structured as follows: Section 2 introduces the new challenges that the JSON data model presents and overviews related research. In Section 3, we introduce the 3 algebra levels. Due to lack of space, levels 1 and 2 are greatly summarized; we expand on level 3 as it is the most relevant for query optimization. Next, in Section 4, we present the main properties of the operators introduced in the previous Section. Finally, we close the paper with an overview of future research.

2. Background and Related Research

Even though there are a few differences in notation and some ancillary aspects, the vast majority of the literature treats JSON data as an edge-labeled tree structure, with some constraints imposed by the JSON standard[1]. The labels of the outgoing edges of a document node are called *keys*, and each subtree represents a *value*. For leaves, the value is an atomic value assigned to it; for internal nodes, the value is either a document or an array. Moreover, all internal nodes are of *document* type or of `array` type. For a document node, all outgoing edges must be labeled by distinct strings; For an array node, the outgoing edges are labeled by "1", "2", "3", ...and they are considered

[1]https://www.json.org/json-en.html

DOLAP 2024: 26th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data, co-located with EDBT/ICDT 2024, March 25, 2024, Paestum, Italy

tlllano01@louisville.edu (T. LLano-Rios); khalefam@oldwestbury.edu (M. Khalefa); abadia@louisville.edu (A. Badia)

ordered. Finally, the root node is always of document type. Thus, a JSON document means a document/tree that respects all the constraints listed here.

Most JSON databases deal with *collections* of documents. The definition of 'collection' is not part of the JSON standard, and no universal definition seems to exist. However, most existing systems treat collections as multisets, that is, groups of documents with no order and where repetitions are allowed. We will follow this practice.

Existing approaches to handling JSON data, like using Nested Relational Algebra (NRA) or XML query languages like XQuery, are not fully appropriate for JSON, as JSON is different from nested relations or XML. In the nested relational model, the set constructor and the tuple constructor strictly alternate [3], while in JSON, arrays (the closest construct to nested relations) may contain documents, atomic values or other arrays. Moreover, nested relations have a fixed schema, and all tuples in a relation must follow the schema. XML and JSON, on the other hand, are semi-structured; but they are different from each other in several aspects. First, in XML all data is ordered (although most query languages for XML do not take advantage of this order), while JSON mixes ordered and unordered data. Second, JSON is structurally deterministic (in a tree representing a JSON document, each path from a leaf to the root is unique), while XML, is non-deterministic [1]. Related to this, JSON deals with groups of related items by using arrays, while XML, which does not have arrays, deals with groups of related items by repeating tags.

In spite of the differences, some systems use NRA to deal with JSON data, for instance, the Proteus system [4]. However, the lack of a fixed schema leads to problems. In the nested relational model, we can assign a schema to a nested relational table R, $schema(R)$. The definition of the operators in NRA depends on this schema. For instance, the projection $\pi_{A_1,\ldots,A_n}(R)$ is well-defined only if $\{A_1,\ldots,A_n\} \subseteq schema(R)$. But given a collection $\mathcal{C}$, it can be the case that $\{A_1,\ldots,A_n\}$ are not attributes of document d for some $d \in C$ -even as they are in other documents. One must decide what it means to project on $\{A_1,\ldots,A_n\}$ when some of these attributes are not present -in the extreme case (when no such attribute is present), should we return the empty document? In the case of projecting inside arrays, NRA offers no guidance. Other NRA operators are also affected by this problem. The *nest* operator is traditionally denoted by $\nu_{B(A_1,\ldots,A_n)}(R)$, where $B \notin schema(R)$ and $A_1,\ldots,A_n \in schema(R)$. Intuitively, R is 'grouped by' all attributes in $schema(R) - \{A_1,\ldots,A_n\}$, and on each group, the values of attributes $A_1,\ldots,A_n$ form a nested relation called B. Clearly, this definition does not work when different documents may have different attributes 'outside' of $A_1,\ldots,A_n$ -or not all of $A_1,\ldots,A_n$ may be present. Similar problems exist with the *unnest* operator. This in turn causes uncertainty as to which properties of NRA hold/do not hold when dealing with JSON. For instance, the unnest of the nest of a relation R is supposed to yield R back, but this may not be the case with JSON data: assume a collection with documents

```
{a: 1, b: "foo"}, {a: 1}
```

If we nest by attribute 'a', in most systems we would get

```
{a: 1, b:["foo"]}
```

Since the second document does not contribute to the result, it is lost; no version of unnest will allow us to recover the original collection.

Most past research on query languages for XML has focused on supporting XPath and XQuery. This has influenced the development of approaches like JSONPath [5, 6]. Document query languages need paths but tend to not navigate across axes like XPath. Research effort [1], which adapts elements of XPath to work on JSON data, is a step in the right direction, but we believe it is only part of the answer. One must also have operators that work on collections of documents. In this sense, our work can be considered complementary to XPath-based approaches [5, 6, 1]: the navigation capabilities presented there could be incorporated into the operators of our language, especially the *selection* operator (see next section).[2]

The rapid development of systems for JSON data has led to a proliferation of different query languages -in most cases, without formal semantics. For instance, MongoDB and CouchDB use their own query languages, defined informally ([2]). Query optimization on these systems is mostly rule-based, with limited use of cost-based considerations ([7, 8]). A recent proposal, SQL++ ([9]), has been adopted by CouchBase and Asterix. SQL++ offers a great deal of flexibility, as several issues that arise with JSON data are treated as parameters that can be set by the user. However, SQL++ assumes a data model that is a superset of JSON, and it also lacks formal semantics. No query optimization for SQL++ has been developed so far. Jaql ([10]) is a scripting system designed to run on Hadoop. Queries are expressed as sequences of statements, and query optimization is rule-based and guided by heuristics. RumbleDB ([11]) is a system to process JSON queries using the JSONiq query language. JSONiq [12] provides a declarative query language based on XQUERY FLWOR expressions. J-Logic[13] gives a query language based on non-recursive Datalog. However, this work does not discuss query optimization. The difficulty of querying heterogeneous documents, which may cause the user to write a query with incorrect paths, has motivated research to correct such paths using information about the data ([14, 15]).

[2]It is notable that [1] focuses the analysis of MongoDB's `find` operator, while our approach is closer to the more general aggregation pipeline.

3. A Document Algebra

We adopt the standard definition of JSON data, and see a JSON document as an edge-labeled tree. We use d, d', …as variables over documents. In the language, paths in the document/tree are denoted by sequences of labels (denoting the labels of the edges). Of course, an arbitrary sequence of labels may not exist in a given tree; in such cases, we say the path is *not realized* in the document. Complete paths (from the root node of the document to a leaf) are called *attributes*. We use $p, p_1, \ldots, p_n$ as variables over paths, and P, P' as variables over sets of paths. The function $Eval(p, d)$ provides the value of path p in a document d. When p is not realized in d, $Eval(p, d)$ returns the special value `undefined`. When p is realized, $Eval(p, d)$ returns the subtree rooted at the node where the path ends, for internal nodes (note that this may be a document or an array); when p is an attribute (complete path), $Eval(p, d)$ returns the atomic value attached to the leaf (this may include `null`). The schema of a document d, in symbols $schema(d)$, is the set of all attributes in the document. A collection, denoted by $\langle , \rangle$, is treated as a multiset of JSON documents. We use $\mathcal{C}$, $\mathcal{C}_1$, etc. as variables over collections. Since documents in a collection may have different schemas, we also define:

$$cover(\mathcal{C}) = \bigcup_{d \in \mathcal{C}} schema(d)$$

When comparing two documents (especially in section 4) we use *weak equality*, which ignores order within arrays, i.e. two arrays are considered equal if they contain the same values, even in a different order.

We now describe the three levels of the algebra. Due to lack of space, for levels 1 and 2 we only describe operators at an intuitive level and mention only features that are particular to JSON data. We provide a full definition of level 3 operators, which are the focus of this paper.[3]

3.1. First Level: A JSON Datatype

The first level focuses on operators that work on individual documents. This level allows us to define JSON documents as an *abstract data type*, independent from the representation of data, be it BISON, Mison [16], Pison [17] or any other schema. The operators are:

- *projection*: given a set of paths P, document d, the projection of P from d, denoted $\pi_P(d)$, takes the subtree of d formed by all paths with a prefix in P. Note the following, non-standard behavior:
  - if some paths are realized and others are not, the projection picks the paths that are realized in the document and ignores the rest. In particular, if no path is realized in the document ($P \cap schema(d) = \emptyset$), the empty document (denoted $\{\}$) is returned.
  - This works for complete and incomplete paths. For instance, in $d = \{a : 1, b : \{c : 2, e : 3\}\}$, $\pi_b(d) = \{b : \{c : 2, e : 3\}\}$; $\pi_{b.c}(d) = \{b : \{c : 2\}\}$. Also, $\pi_{a.g}(d) = \pi_f(d) = \emptyset$. In particular, we allow picking (parts of) values from within an array.
- *merge(d_1, d_2)*, where d_1 and d_2 are JSON documents, is an operation that 'puts together' both documents under a common root. For instance,
  merge({a: 1, b:{c:2, d:3}}, {e: 4, b: [5, 6]}) = {a: 1, b:{c:2, d:3}, e: 4, b: [5, 6]}
  In particular, $merge(d, \{\}) = merge(\{\}, d) = d$.
  Note that have a technical issue in this operator: if the same path exists in both d_1 and d_2[4], the result is not a legal JSON document. The situation is similar to that of (nested) relational algebra and Cartesian product, so we adopt the same solution: we assume a *rename* operator allows us to change labels in a path as needed.
- $\mu_p(d)$ (called the *unnest* of d by p), where d is a JSON document and p is a path. This operator returns a collection, i.e. a multiset of documents, defined as follows: if $Eval(p, d)$ is an array, for each value i in the array, we create a document d_i composed of: all of d except the array p, and the single value i. If $Eval(p, d)$ is not an array, $\mu_P(d) = \langle d \rangle$.
- $\sigma_\alpha(d)$, where d is a JSON document and α is a condition. Conditions are defined as usual: if p, p_1, p_2 are paths, c a constant, and θ a comparison operator, $p\ \theta\ c$, $p_1\ \theta\ p_2$, $\exists p$ are conditions. In addition, conjunction, disjunction, and negation of conditions are also conditions. $\sigma_\alpha(d)$ returns one of 'true', 'false', 'unknown' or 'undefined' by evaluating α in d. The evaluation process is defined as usual; 'undefined' is returned when a path used in α is not realized in d and 'unknown' when a path used in α evaluates to null. It is here that other approaches like JSONPath ([6]) could be incorporated into the language.

3.2. Second Level: Collection Manipulation

The second level provides basic operations on collections. The operators include *project*, *select*, *nest*, *unnest*, *Cartesian product* and *union*, *intersection* and *difference*, suitably modified from their usual meanings in (nested) relational algebra.[5] As in level 1, we only provide a summary description.

[3]A full detailed description of all algebra levels is available as a Technical Report from the authors.

[4]In fact, it's enough that a path in d_1 and a path in d_2 start with the same label.

[5]A full description of this level (with some added operators like *aggregate* and *order*, which support typical query capabilities) is in the Technical Report.

- The *selection* operator, written $\sigma_\alpha(\mathcal{C})$, takes as input a collection $\mathcal{C}$ and as parameter a *condition* α, defined as above, and picks documents within $\mathcal{C}$ that satisfy the condition (with slight abuse of notation, we use σ for this operator, as context will help make clear which selection we are using). By rejecting documents where the condition evaluates to 'undefined' or 'unknown', we are following SQL semantics –obviously, this could be changed if desired.

$$\sigma_\alpha(\mathcal{C}) = \langle d \in \mathcal{C} \mid \sigma_\alpha(d) = true \rangle$$

- The *projection* operator $\pi_P(\mathcal{C})$ takes as input a collection $\mathcal{C}$ and as a parameter a set of path expressions P. Again, we abuse the notation to use the same symbol as the previously defined projection. While a projection on an individual document may return an empty document, the projection on collections returns non-empty documents only. Not all document systems follow this semantics;[6] but this choice makes some properties that follow simpler and cleaner.

$$\pi_P(\mathcal{C}) = \langle \pi_P(d) \mid d \in \mathcal{C} \wedge \pi_P(d) \neq \{\} \rangle$$

- The *Cartesian product* operator takes as input two collections $\mathcal{C}_1$ and $\mathcal{C}_2$:

$$\mathcal{C}_1 \times \mathcal{C}_2 = \langle merge(d_1, d_2) \mid d_1 \in \mathcal{C}_1, d_2 \in \mathcal{C}_2 \rangle$$

  Note that we assume renaming as needed. As in the case of the traditional operator, if either collection is empty, the product returns the empty collection.
- the *unnest* operator takes as input a collection $\mathcal{C}$ and as parameter a path expression p. As in the case of selection and projection, we reuse the symbol for document unnest.

$$\mu_p(\mathcal{C}) = \bigcup_{d \in \mathcal{C}} \mu_p(d)$$

  Note that we use multiset union.
- the *nest* operator, $\nu_{p:p_1,\ldots,p_n}(\mathcal{C})$, takes as input a collection $\mathcal{C}$ and as parameter the path expressions $p : p_1, \ldots, p_n$. Unlike nesting in NRA, here $p_1, \ldots, p_n$ denote the nesting (common) attributes, and p, where we want to place the array created by gathering the nested (pushed down) attributes -which, in any document d, are all attributes not among $p_1, \ldots, p_n$. To form groups, all documents d that have the same values for all paths in $p_1, \ldots, p_n$ are gathered. Here, 'same values' includes the case where some paths are not realized in a document, so that their values are undefined. For the purposes of nesting, nulls are all equal, and so are 'undefined'.[7] For instance, if we nest by paths p_1 and p_2, then all documents where both paths are null form a group; all documents where both paths are not realized generate another group; all documents where p_1 is not realized, p_1 is realized and has the same value, form another group, and so on.
  To accommodate group by/aggregate queries, we further specify what is collected under p: for an aggregate function, we need to specify which function and to which attribute it applies. We assume, when no aggregate is specified, that the rest of the document (all but $p_1, \ldots, p_n$) is gathered as a value of the array p. When a document has no 'rest', an empty document is deposited in the array. Finally, we use a special aggregate function 'push' that concatenates arrays, i.e. *push([2, {a: 1}], ["foo", 4]) = [2, {a: 1}, "foo", 4].*
- the *union, intersection and difference* operators each take as input two collections $\mathcal{C}_1$ and $\mathcal{C}_2$. These operators work exactly as in the traditional case for multisets; for instance, the union $\mathcal{C}_1 \cup \mathcal{C}_2$ is the collection of documents in either $\mathcal{C}_1$ or $\mathcal{C}_2$, repeated as many times as the sum of their appearances; similarly for intersection $\mathcal{C}_1 \cap \mathcal{C}_2$ and difference $\mathcal{C}_1 - \mathcal{C}_2$. Note that, in the definition of intersection and set difference, 'weak' equality (where the order within arrays is ignored) is used to compare documents.[8]

3.3. Third Level: Macro-operators

The third algebra level contains operators which could be expressed using the 'basic' ones, but that we want in our algebra for optimization reasons. A typical example of this is the (relational) join. We believe that in document algebra, several operations are similar to join in this respect. Another important reason to have a separate level for 'basic' operations is that having a short list of simple operators makes finding and proving equivalences much easier.

The main additional operators are:

- *join*: as in the relational case, a join is a selection that follows a Cartesian product. Defining join in this manner settles what is meant exactly by a join with complex elements, regardless of whether the comparison involves top-level attributes or nested ones.[9]

$$\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2 = \sigma_\alpha(\mathcal{C}_1 \times \mathcal{C}_2)$$

- *semijoin*, *outerjoin*, and *antijoin*: once joins are defined, we can also define these closely related operators. An important difference with the relational case is that documents in a collection without a match in the other are simply left as they are in the results -due to the lack of schema, no padding is necessary. Whenever

[6]For instance, MongoDB allows returning empty documents.

[7]In the absence of a standard, we follow SQL semantics. An alternative is to let the user decide how to deal with such cases, as is done in [9]; however, such approaches make it very hard to prove properties.

[8]As in the case of Relational Algebra, not all three set operators are needed; we include all three at this level for simplicity.

[9]An issue that may introduce ambiguity in versions of NRA ([18]).

$cover(C_1) \cap cover(C_2) = \emptyset$, the definition of such operators is similar to that of (flat, nested) relational algebra; otherwise, a problem arises that requires careful renaming. In order to avoid this, here we provide a non-standard definition that avoids the problem:

(left semijoin) $\mathcal{C}_1 \ltimes_\alpha \mathcal{C}_2 = \{d \in \mathcal{C}_1 \mid \{d\} \bowtie_\alpha \mathcal{C}_2 \neq \emptyset\}$

(left antijoin) $\mathcal{C}_1 \rhd_\alpha \mathcal{C}_2 = \{d \in \mathcal{C}_1 \mid \{d\} \bowtie_\alpha \mathcal{C}_2 = \emptyset\}$

(left outerjoin) $\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2 = (\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) \cup (\mathcal{C}_1 \rhd_\alpha \mathcal{C}_2)$

- *nest-join* (NJ) and *nest-outerjoin* (NOJ): when there exists a 1-M relationship between two entities E_1 and E_2, document databases offer the possibility of storing the data using embedding (in which case each document in E_1 contains an array of E_2 sub-documents) or in two different collections, one for E_1 and another for E_2, with some attributes serving as 'foreign keys' to maintain the links among documents. In the latter case, a join will put together documents in E_1 with related documents in E_2, and a nesting by (the attributes of) E_1 will result in a structure similar to the embedding. The nest-join is an operation that combines both the join and nesting in a single step; similarly, we define *nest outerjoin* The goal here is to provide a target for optimization, since implementing this operator with a single algorithm in a single pass over the data can lead to improved plans.[10] Note the similarity with ideas proposed in literature under names like *groupjoin* ([19]).

$$NJ_{p,c}(C_1, C_2) = \nu_{p:cover(\mathcal{C}_1)}(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2)$$

The *nest-outerjoin* operator is defined in the obvious similar manner:

$$NOJ_{p,c}(C_1, C_2) = \nu_{p:cover(\mathcal{C}_1)}(\mathcal{C}_1 ⟕_\alpha \mathcal{C}_2)$$

- *narrow selection*: when evaluating a condition involving a single value against an array in document d, we offer operators *all* and *some*, similar to those of SQL, to specify how the condition is to be interpreted. The final result of such a comparison, however, is a single decision on whether d qualifies for the answer. But what is sometimes wanted is the document d, but with only those elements in the array that make the condition true (in other words, the array is filtered). A narrow selection $\sigma'_\alpha(\mathcal{C})$, where α is a condition as just described, does exactly this. For instance, in document `{a:1, b:[2, 3, 4, 5]}`, narrow selection $\sigma'_{b>3}$ returns[11] document `{a:1, b:[4, 5]}`. It is not hard to see that σ' can be defined in terms of basic operators (first, unnest the input; then apply a (regular) selection; finally, nest the result):

$$\sigma'_\alpha(\mathcal{C}) = \mu_p(\sigma_\alpha(\nu_p(\mathcal{C})))$$

Note that if no element in the array makes the condition true, the whole document is erased from the result.

- *extended nest*: sometimes we may want to nest documents creating several arrays. For instance, in a collection with documents `{a:1, b:2, c:3}`, `{a:1, b:4, c:5}` we may want to nest by 'a' and separate the 'b' and 'c' elements, to create result `{a:1, b:[2, 4], c:[3, 5]}`. The extended nest operator allows the specification of more than one nested component, to achieve exactly this result. It would seem that this can be achieved with regular nesting, by first projecting the input collection into `{a, b}` and nesting the result by 'a'; projecting again the input into `{a, c}` and again nesting the result by 'a'; and finally joining the two previous results:

$$E\nu_{p,q,r}(\mathcal{C}) = \nu_{p1:p}(\pi_{p,q}(\mathcal{C})) \bowtie_{p1=p2} \nu_{p2:p}(\pi_{p,r}(\mathcal{C}))$$

However, for this to work in general, we need to assume *order* -which, up to now, we have disregarded, even inside arrays. To see why, assume collection with documents $\langle\{a : 1, b : 2, c : 3\}, \{a : 1, b : 4\}, \{a : 1, c : 5\}\rangle$. If we extend nest all documents by a, collecting b and α , we want to get the document $\{a : 1, b : [2, 4, \{\}], c : [3, \{\}, 5]\}$, not $\{a : 1, b : [2, 4], c : [3, 5]\}$ (among other reasons, it's the only way that extended nest and extended unnest will be inverses of each other). Note that nest will generate empty documents to mark where values of `b` or `c` are not present, but the resulting arrays need to be 'paired up' consistently. Thus, in the definition of this operator, we assume that nesting has generated an ordered array, and has done so using the same order in both nestings.
Note that, for this operator, we require the user to specify 'grouping' attributes but also specific 'grouped' attributes to be collected in the array. It can be shown that this definition works for an arbitrary number of arrays.

- *extended unnest*: similarly, the unnest operator is extended to maintain symmetry. Note that a regular unnest, applied twice to the previous result (document `{a:1, b:[2, 4], c:[3, 5]}`) in any order (even if applied separately to projections of the result and then combined) would yield not the original collection, but one with 4 documents (with all combinations of values in 'b' and 'c'). Thus, the purpose of the extended unnest is to have an operation that undoes exactly what the extended nest does: the extended unnest of `{a:1, b:[2, 4], c:[3, 5]}` yields collection ⟨ `{a:1, b:2, c:3}, {a:1, b:4, c:5}` ⟩.

$$E\mu_{p,q,r}(\mathcal{C}) = (\mu_q(\pi_{p,q}(\mathcal{C}))) \bowtie_p (\mu_r(\pi_{p,r}(\mathcal{C})))$$

Note that both q and r must be paths to arrays.

[10]In MongoDB's aggregate pipeline, only this operator (called `lookup`) is supported; join and outerjoin are not.
[11]Paths with arrays in a condition have an *existential* reading.

The reason to define these operators is that it is clear that a direct implementation can yield much better performance than a rewriting. Consider, for instance, the narrow selection as a one-pass operation on a collection versus its rewriting. Also, the optimized implementation of the extended nest could easily enforce the same order for array formation, as it could do one-pass sorting/hashing over the collection.

Other extended operators can be defined, for instance, to capture similar capabilities in SQL –like window aggregators or an extension matching the CASE statement. On top of this, extended operators proposed for the relational case may make sense here –for instance, the θ MDA (Multi-Dimensional Aggregation) of [20].

4. Properties

First, we observe that the second level of the algebra (and, by extension, the third) have *closure*: all operators take in collections of JSON documents and produce as output collections of JSON documents. This does not apply to the first-level operators; however, these operators are not meant to be composed but used as 'auxiliary methods' to manipulate individual documents.

To support query optimization, next we list properties of individual operators, both in the core (second-level) and the extended (third-level) of the algebra.[12]

4.1. Properties of Basic Operators

4.1.1. Projection

First, we observe that our projection does not remove duplicates. In our algebra, duplicate removal is accomplished by using nesting without any aggregate or gathering of results in an array a case denoted by $\nu_{\emptyset:p_1,\ldots,p_n}(\mathcal{C})$. This simply returns one document for any repeated combination of values of $p_1,\ldots,p_n$ in $\mathcal{C}$. However, the nest as defined will produce an empty document if there exists one or more documents in the input where none of $p_1,\ldots,p_n$ exists. To solve this, we define

$$distinct_{p_1,\ldots,p_n}(\mathcal{C}) = \{d \in \nu_{\emptyset:p_1,\ldots,p_n}(\mathcal{C}) \mid d \neq \{\}\}$$

It follows from this that projecting after a nest can be done simply by nesting on the projected attributes.

Lemma 4.1. *If P, P' are sets of attributes with $P \subseteq P'$, then $\pi_P(\nu_{\emptyset:P'}(\mathcal{C})) = distinct_P(\mathcal{C})$.*

For instance, if we have $\pi_a(\nu_{p:a,b}(C))$ on collection $\langle\{a:1, b:2, c:3\}, \{a:1, b:2, c:4\}\rangle$, the result is $\{a:1\}$, which we could obtain by simply doing a grouping by a.

[12]Unfortunately, due to the lack of space, all proofs are omitted, and will be available on the Technical Report.

Another basic observation is that if a projection uses paths that are not realized in any document in a collection, such paths can be ignored.[13] On the other hand, at least one path must be present for the projection to consider a document.

Lemma 4.2. *For any set of paths $P = \{p_1,\ldots,p_n\}$, collection $\mathcal{C}$,*

- $\pi_P(\mathcal{C}) = \pi_{P\cap cover(\mathcal{C})}(\mathcal{C})$
- $\pi_P(\mathcal{C}) = \pi_P(\sigma_{\exists p_1 \vee \ldots \vee \exists p_n}(\mathcal{C}))$

The basic properties of projection are given in the next lemma.

Lemma 4.3. *For any set of paths P, P_1, P_2, collection $\mathcal{C}$,*

- $\pi_P(\pi_P(\mathcal{C})) = \pi_P(\mathcal{C})$.
- *If* $P_1 \subseteq P_2$, $\pi_{P_1}(\pi_{P_2}(\mathcal{C})) = \pi_{P_1}(\mathcal{C})$.

The behavior of projection with other operators is described by the next lemma.

Lemma 4.4. *For any set of paths P, condition α,*

- $\pi_P(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) = \pi_{P_1}(\mathcal{C}_1) \bowtie_\alpha \pi_{P_2}\mathcal{C}_2$, *where* $P_1 = P \cap cover(\mathcal{C}_1)$ *and* $P_2 = P \cap cover(\mathcal{C}_2)$.
- $\pi_P(\mathcal{C}_1 \cup \mathcal{C}_2) = \pi_P(\mathcal{C}_1) \cup \pi_P(\mathcal{C}_2)$.

Note that $\pi_P(\mathcal{C}_1 \cap \mathcal{C}_2) \neq \pi_P(\mathcal{C}_1) \cap \pi_P(\mathcal{C}_2)$, as there may be $d_1 \in \mathcal{C}_1$, $d_2 \in \mathcal{C}_2$, $d_1 \neq d_2$ such that $\pi_P(d_1) = \pi_P(d_2)$. For the same reason, $\pi_P(\mathcal{C}_1 - \mathcal{C}_2) \neq \pi_P(\mathcal{C}_1) - \pi_P(\mathcal{C}_2)$. However, a weaker property holds:
$\pi_P(\mathcal{C}_1 \cap \mathcal{C}_2) \subseteq \pi_P(\mathcal{C}_1) \cap \pi_P(\mathcal{C}_2)$.

The behavior of projection in combination with nest and unnest is more complex, due to the issue of duplicates. To illustrate, the unnest of document $\{a:1, b:[2,3]\}$ would create two documents, $\{a:1, b:2\}$ and $\{a:1, b:3\}$, so that a projection on a would need to return two copies of $\{a:1\}$ -hence, it cannot be pushed down. As for nesting, the opposite effect happens: on the grouping attributes, nesting removes duplicates. For instance, projecting on a over the collection $\langle\{a:1, b:2\}, \{a:1, b:3\}\rangle$ would return two copies of $\{a:1\}$; after a nesting by a, the projection would return only one copy. Note that both transformations would work for systems where projection removes duplicates; hence, we have the following.

Lemma 4.5. *For any set of paths P, collection $\mathcal{C}$,*

- *if* $p \notin P$, $distinct_P(\mu_p(\mathcal{C}) = \pi_P(\mathcal{C})$
- *if* $p \in P$, $\pi_P(\nu_{p:p}(\mathcal{C})) = distinct_P(\mathcal{C})$

[13]Note that P is set of paths and $cover(\mathcal{C})$ a set of attributes (complete paths); in the lemmas, their intersection denotes the set of $p \in P$ such that p is a prefix of some attribute in $cover(\mathcal{C})$.

4.1.2. Selection

First, we define an auxiliary concept that will be useful.
Definition 4.6. *Let α be a condition, then $Pt(\alpha)$ is the set of paths mentioned in α:*

- $Pt(p_1 \theta p_2) = \{p_1, p_2\}$
- $Pt(p\theta c) = Pt(\exists p) = \{p\}$
- $Pt(\varphi \wedge \psi) = Pt(\varphi \vee \psi) = Pt(\varphi) \cup Pt(\psi)$

Lemma 4.7. *For any conditions α_1 , α_2,*

- $\sigma_{\alpha_1}(\sigma_{\alpha_2}(\mathcal{C})) = \sigma_{\alpha_2}(\sigma_{\alpha_1}(\mathcal{C}))$
- $\sigma_{\alpha_1 \wedge \alpha_2}(\mathcal{C}) = \sigma_{\alpha_1}(\sigma_{\alpha_2}(\mathcal{C}))$

Lemma 4.8. *Let α, β be conditions, with α expressible as $\alpha_1 \wedge \alpha_2$, $Pt(\alpha_i) \subseteq cover(\mathcal{C}_i)$, $i = 1, 2$, then*

$$\sigma_\alpha(\mathcal{C}_1 \bowtie_\beta \mathcal{C}_2) = (\sigma_{\alpha_1}(\mathcal{C}_1)) \bowtie_\beta (\sigma_{\alpha_2}(\mathcal{C}_2))$$

An immediate consequence of this lemma is that, if $Pt(\alpha) \subseteq cover(\mathcal{C}_1)$, $\sigma_\alpha(\mathcal{C}_1 \bowtie_\beta \mathcal{C}_2) = \sigma_\alpha(\mathcal{C}_1) \bowtie_\beta \mathcal{C}_2$; and if $Pt(\alpha) \subseteq cover(\mathcal{C}_2)$, $\sigma_\alpha(\mathcal{C}_1 \bowtie_\beta \mathcal{C}_2) = \mathcal{C}_1 \bowtie_\beta \sigma_\alpha(\mathcal{C}_2)$.
Lemma 4.9. *For any condition α, we have:*

- $\pi_P(\sigma_\alpha(\mathcal{C})) = \pi_P(\sigma_\alpha(\pi_{P \cup Pt(\alpha)}(\mathcal{C})))$
- $\sigma_\alpha(\mathcal{C}_1 \cup \mathcal{C}_2) = \sigma_\alpha(\mathcal{C}_1) \cup \sigma_\alpha(\mathcal{C}_2)$
- $\sigma_\alpha(\mathcal{C}_1 \cap \mathcal{C}_2) = \sigma_\alpha(\mathcal{C}_1) \cap \sigma_\alpha(\mathcal{C}_2)$
- $\sigma_\alpha(\mathcal{C}_1 - \mathcal{C}_2) = \sigma_\alpha(\mathcal{C}_1) - \mathcal{C}_2 = \sigma_\alpha(\mathcal{C}_1) - \sigma_\alpha(\mathcal{C}_2)$

Lemma 4.10. *Let α be a condition with $Pt(\alpha) \subseteq P$. Then*

$$\sigma_\alpha(\nu_{p:P}(\mathcal{C})) = \nu_{p:P}(\sigma_\alpha(\mathcal{C}))$$

That is, we can push a projection past a nest if all attributes in the condition are among the grouping attributes. This is similar to `HAVING` in relational scenarios: the selection qualifies all tuples in a group, or none.

4.1.3. Union

Union properties are especially important for distributed computation: a collection distributed over the nodes of a cluster can be represented as a union of 'pieces', so when an operator can be 'pushed down' past union, this means that it can be executed in the nodes of the cluster. In this context, lemma 4.12 is the basis for a 'broadband' join.

Lemma 4.11. *For any collections $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$,*

- $\mathcal{C}_1 \cup \mathcal{C}_2 = \mathcal{C}_2 \cup \mathcal{C}_1$.
- $(\mathcal{C}_1 \cup \mathcal{C}_2) \cup \mathcal{C}_3 = \mathcal{C}_1 \cup (\mathcal{C}_2 \cup \mathcal{C}_3)$.

Lemma 4.12. *Let $\mathcal{C}_2 = \bigcup_{i=1}^n \mathcal{C}_2^i$; then*

$$\mathcal{C}_1 \bowtie \mathcal{C}_2 = \mathcal{C}_1 \bowtie \bigcup_{i=1}^n \mathcal{C}_2^i = \bigcup_{i=1}^n (\mathcal{C}_1 \bowtie \mathcal{C}_2^i)$$

We can push unnest past a union:

Lemma 4.13. *For any collections $\mathcal{C}_1, \mathcal{C}_2$, path p,*
$$\mu_p(\mathcal{C}_1 \cup \mathcal{C}_2) = \mu_p(\mathcal{C}_1) \cup \mu_p(\mathcal{C}_2)$$

As for nest, we can decompose the nesting into two steps by using the 'push' aggregate on the second one (note that the first nesting creates arrays, so 'push' will combine such arrays).

Lemma 4.14. *For any collections $\mathcal{C}_1, \mathcal{C}_2$, set of paths P,*
$\nu_{p:P}(\mathcal{C}_1 \cup \mathcal{C}_2) = \nu_{push(p):P}(\nu_{p:P}(\mathcal{C}_1) \cup \nu_{p:P}(\mathcal{C}_2)$

As an example, let $\mathcal{C}$ be the collection
$\langle\{a:1, b:2\}, \{a:1, b:3\}, \{a:1, b:4\}\rangle$
be split into $\mathcal{C}_1 = \langle\{a:1, b:2\}, \{a:1, b:3\}\rangle$ and $\mathcal{C}_2 = \langle\{a:1, b:4\}\rangle$. Then
$\nu_{p:a}(\mathcal{C}) = \langle\{a:1, p:[\{b:2\}, \{b:3\}, \{b:4\}]\}\rangle$;
$\nu_{p:a}(\mathcal{C}_1) = \langle\{a:1, p:[\{b:2\}, \{b:3\}]\}\rangle$;
$\nu_{p:a}(\mathcal{C}_2) = \langle\{a:1, p:[\{b:4\}]\}\rangle$;
$\nu_{push(p):a}(\nu_{p:a}(\mathcal{C}_1), \nu_{p:a}(\mathcal{C}_2)) = \langle\{a:1, p:[\{b:2\}, \{b:3\}, \{b:4\}]\}\rangle$. Note that this idea can be extended to traditional (numerical) aggregates, and is similar to the way many modern distributed systems compute aggregate queries ([21]).

4.1.4. Nest and Unnest

Nest and unnest are hard to combine; it is not true, in general, that $\nu_{p:p_1,\ldots,p_n}(\mu_p(\mathcal{C})) = \mathcal{C}$. To see this, let $\mathcal{C} = \langle\{a:1, b:[2,3]\}, \{a:1, b:[4,5]\}\rangle$. Then the unnest of b will produce a collection $\langle\{a:1, b:2\}, \{a:1, b:3\}, \{a:1, b:4\}\{a:1, b:5\},\rangle$, and the nesting by a will produce $\langle\{a:1, b:[2,3,4,5]\}\rangle$. This is a well-known issue in NRA ([3]). However, it could be the case that unnesting acts as the inverse of nesting. This does not happen in JSON query languages with similar operators (for instance, MongoDB), but the following states that this property holds in our algebra.

Lemma 4.15. *For any paths $p_1, \ldots, p_n$, collection $\mathcal{C}$,*
$$\mu_p(\nu_{p:p_1,\ldots,p_n}(\mathcal{C}) = \mathcal{C}$$

4.2. Properties of Extended Operators

Recall that we use $\bowtie$ for joins, $\sqsupset\!\!\bowtie$ for (left) outerjoin, $\ltimes$ for (left) semijoin and σ' for narrow selection.

Lemma 4.16. *(Joins) Joins are associative and commutative:*
$$(\mathcal{C}_1 \bowtie \mathcal{C}_2) \bowtie \mathcal{C}_3 = \mathcal{C}_1 \bowtie (\mathcal{C}_2 \bowtie \mathcal{C}_3)$$

Lemma 4.17. *(Outerjoins, part 1) Whenever $P \subseteq cover(\mathcal{C}_1)$, for any condition α,*
$$distinct_P(\mathcal{C}_1 \sqsupset\!\!\bowtie_\alpha \mathcal{C}_2) = distinct_P(\mathcal{C}_1)$$

Note that this does not work with projection without duplicate removal, since the outerjoin may introduce duplicates whenever a document in $\mathcal{C}_1$ matches more than one document in $\mathcal{C}_2$.

Following previous work [22], we prove that an outerjoin followed by a *rejecting condition* can be transformed into a (inner) join. We say a condition α is rejecting on attribute set P if $Pt(\alpha) \nsubseteq P$ – since α is guaranteed not to be true when attributes in P are not present on the input.

Lemma 4.18. *(Outerjoins, part 2) Let α be a condition with $Pt(\alpha) \subseteq cover(\mathcal{C}_2)$, β an arbitrary condition, then*
$$\sigma_\alpha(\mathcal{C}_1 ⟖_\beta \mathcal{C}_2) = \mathcal{C}_1 \bowtie_\beta \mathcal{C}_2$$

That is, the outer join becomes a join when a selection involves attributes from the right side only -as this implies that it 'rejects absents' on the left side. Technically, this holds only if $cover(\mathcal{C}_1) \cap cover(\mathcal{C}_2) = \emptyset$; we assume renaming when this is not the case.

Lemma 4.19. *(Narrow Select, Part I) Narrow select and unnest commute: for any condition α,*
$$\mu_P(\sigma'_\alpha(\mathcal{C})) = \sigma_{Fl_p(\alpha)}(\mu_P(\mathcal{C}))$$
Narrow select and nest also commute:
$$\sigma'_\alpha(\nu_{p:p_1,\ldots,p_n}(\mathcal{C})) = \nu_p(\sigma_{Fl_p(\alpha)}(\mathcal{C}))$$

As an example of the first statement, given collection $\mathcal{C}$ with single document $\{a : 1, b : [2, 3, 4]\}$, $\sigma'_{b \geq 3}(\mathcal{C})) = \langle\{a : 1, b : [3, 4]\}\rangle$; and unnesting this yields $\langle\{a : 1, b : 3\}, \{a : 1, b : 4\}\rangle$. If we apply the unnest first to $\mathcal{C}$, we get the collection $\langle\{a : 1, b : 2\}, \{a : 1, b : 3\}, \{a : 1, b : 4\}\rangle$. If we start with this collection and run the same operators, we have an example of the second property.

For the following property, we first need an additional definition: given condition α , path p, we say that α is *p-centered* if all paths in $Pt(\alpha)$ have p as a prefix. Then, the flattening of α (in symbols, $Fl_p(\alpha)$) for p-centered α , is defined as the condition obtained by substituting all paths $p.q \in Pt(\alpha)$ by q (i.e. by removing the prefix p from all paths in α). Constants and operators remain the same.

Lemma 4.20. *(Narrow Select, Part II) The narrow selection of a nest-join is the same as the nest-join with a regular select pushed down: let α be a condition with $Pt(Fl_p(\alpha)) \subseteq cover(\mathcal{C}_2)$, then*
$$\sigma'_\alpha(\mathcal{C}_1 NJ_{p,\alpha} \mathcal{C}_2) = \mathcal{C}_1 NJ_{p,\alpha} \sigma_{Fl_p(\alpha)}(\mathcal{C}_2)$$
The same holds true for the nest-outerjoin.

As an example, let $d_1 = \{a : 1, b : 2\} \in \mathcal{C}_1$, $d_2 = \{e : 1, f : 4\}$ and $d_3 = \{e : 1, f : 5\}$ both in $\mathcal{C}_2$; a (left) nest-join of $\mathcal{C}_1$ and $\mathcal{C}_2$ on condition $a = e$ will contain the document $\{a : 1, b : 2, e : 1, p : [\{f : 4\}, \{f : 5\}]\}$. A narrow select on $p.f > 4$ will filter the first element of the array, yielding $\{a : 1, b : 2, e : 1, p : [\{f : 5\}]\}$. However, if we use a regular selection with condition $f > 4$ on $\mathcal{C}_2$, we eliminate d_3, and the (left) nest-join would combine d_1 and d_2 to produce the same result.

Selection (regular and narrow) can be also pushed past extended nest and extended unnest, under the right conditions. Extended nest divides the cover of the input collection into 3 parts: a 'flat' part r, an array p and an array q. If the condition attributes are all in r, then the selection can be pushed down. If the condition attributes are in p, then the selection cannot be pushed down as it may also affect q (and the same for condition attributes in q). As an example, in a collection with two documents `{a:1, b:2, c:5}, {a:1, b:4, c:6}` grouping by a, collecting b and c, we get
`{a:1, b:[2, 4], c:[5, 6]}` A selection on a affects the whole group, so either both documents in the input pass it or don't. A selection on b, on the other hand, will take out the accompanying values of c. For instance, selecting $b > 3$ in the input collections takes out the first document, and with it the value 5 of c. But a narrow-select on the output collection wouldn't take out the c value. A similar argument can be made with select and unnest. The following lemma formalizes this:

Lemma 4.21. *Let α be a condition with $Pt(\alpha) \subseteq P$. Then*
$$\sigma_\alpha(enest_{P,Q,R}(\mathcal{C})) = enest_{P,Q,R}(\sigma_\alpha(\mathcal{C}))$$
$$\sigma_\alpha(eunnest_{P,Q,R}(\mathcal{C})) = eunnest_{P,Q,R}(\sigma_\alpha(\mathcal{C}))$$
The same holds using σ'_α instead of σ_α.

Finally, we check that extended nest and unnest are 'well-behaved' extensions of nest and unnest, in the following sense:

Lemma 4.22. *Extended unnest and extended nest are inverses of each other, similarly to unnest and nest:*
$$enest_{P,Q,R}(eunnest_{P,Q,R}(\mathcal{C})) = \mathcal{C}$$
$$eunnest_{P,Q,R}(enest_{P,Q,R}(\mathcal{C})) = \mathcal{C}$$

This is enabled by the caveat, noted when defining the extended nest and extended unnest, that there is an assumption that order is used to make sure that elements inside the arrays are matched.

5. Conclusion and Further Work

We have introduced an algebra for JSON documents with all operators well-defined in the presence of heterogeneous documents. We have designed the algebra in 3 levels: a first one with operators to manipulate single documents; a second one with operators that manipulate collections of documents; and a third one also at the collection level but with derived operators, which are not strictly necessary but offer opportunities for optimization. We have shown algebraic properties that can be used to optimize queries. Our work is independent of physical implementation details (JSON formats, algorithms) and therefore could be adapted by different systems, as far as they offer a query language that can be translated into our algebra. We are currently developing a cost-based query optimization framework for JSON data, centered around the proposed document algebra. As part of this effort, we will show that the core aspects of existing query languages like MongoDB's aggregate pipeline and SQL++ can be translated into our algebra and that doing so offers opportunities for cost-based optimization.

References

[1] P. Bourhis, J. L. Reutter, D. Vrgoč, Json: Data model and query languages, Information Systems 89 (2020) 101478. URL: https://www.sciencedirect.com/science/article/pii/S0306437919305307. doi:https://doi.org/10.1016/j.is.2019.101478.

[2] E. Botoeva, D. Calvanese, B. Cogrel, G. Xiao, Expressivity and complexity of mongodb queries, in: Proc. of the 21st Int. Conf. on Database Theory (ICDT 2018), volume 98 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, pp. 9:1–9:23. doi:10.4230/LIPIcs.ICDT.2018.9.

[3] J. Paredaens, P. De Bra, M. Gyssens, D. Van Gucht, The Nested Relational Database Model, Springer Berlin Heidelberg, Berlin, Heidelberg, 1989, pp. 177–201. URL: https://doi.org/10.1007/978-3-642-69956-6_7. doi:10.1007/978-3-642-69956-6_7.

[4] M. Karpathiotakis, I. Alagiannis, A. Ailamaki, Fast queries over heterogeneous data through engine customization, Proceedings of the VLDB Endowment 9 (2016) 972–983. doi:10.14778/2994509.2994516.

[5] L. Jiang, X. Sun, U. Farooq, Z. Zhao, Scalable processing of contemporary semi-structured data on commodity parallel processors - a compilation-based approach, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 79–92. URL: https://doi.org/10.1145/3297858.3304008. doi:10.1145/3297858.3304008.

[6] S. Goessner, Jsonpath: Xpath for json, https://goessner.net/articles/JsonPath/, 2021.

[7] T. F. Llano-Rios, M. Khalefa, A. Badia, Experimental comparison of relational and nosql document systems: the case of decision support, in: Proc. of the Twelfth TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC), 2020.

[8] T. F. Llano-Rios, M. Khalefa, A. Badia, Evaluation nosql systems for decision support: An experimental approach, in: Proc. of the IEEE Conference on Big Data, 2020, pp. 2802–2811.

[9] K. W. Ong, Y. Papakonstantinou, R. Vernoux, The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases, CoRR abs/1405.3631 (2014). URL: http://arxiv.org/abs/1405.3631. arXiv:1405.3631.

[10] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, E. J. Shekita, Jaql: A scripting language for large scale semistructured data analysis, Proc. VLDB Endow. 4 (2011) 1272–1283. URL: https://doi.org/10.14778/3402755.3402761. doi:10.14778/3402755.3402761.

[11] I. Müller, G. Fourny, S. Irimescu, C. Berker Cikis, G. Alonso, Rumble: data independence for large messy data sets, Proceedings of the VLDB Endowment 14 (2020) 498–506.

[12] D. Florescu, G. Fourny, Jsoniq: The history of a query language, IEEE Internet Computing 17 (2013) 86–90. doi:10.1109/MIC.2013.97.

[13] J. Hidders, J. Paredaens, J. V. den Bussche, J-logic: a logic for querying JSON, CoRR abs/2006.04277 (2020). URL: https://arxiv.org/abs/2006.04277. arXiv:2006.04277.

[14] H. Ben Hamadou, F. Ghozzi, A. Péninou, O. Teste, Schema-independent querying for heterogeneous collections in nosql document stores, Information Systems 85 (2019) 48–67. URL: https://www.sciencedirect.com/science/article/pii/S0306437918302990. doi:https://doi.org/10.1016/j.is.2019.04.005.

[15] O. Rodriguez, F. Ulliana, M.-L. Mugnier, Scalable reasoning on document stores via instance-aware query rewriting, Proc. VLDB Endow. 16 (2023) 2699–2713. URL: https://doi.org/10.14778/3611479.3611481. doi:10.14778/3611479.3611481.

[16] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, D. Kossmann, Mison: A fast json parser for data analytics, Proc. VLDB Endow. 10 (2017) 1118–1129. URL: https://doi.org/10.14778/3115404.3115416. doi:10.14778/3115404.3115416.

[17] L. Jiang, J. Qiu, Z. Zhao, Scalable structural index construction for json analytics, Proc. VLDB Endow. 14 (2020) 694–707. URL: https://doi.org/10.14778/3436905.3436926. doi:10.14778/3436905.3436926.

[18] H.-C. Liu, J. X. Yu, Algebraic equivalences of nested relational operators, Information Systems 30 (2005) 167–204. URL: https://www.sciencedirect.com/science/article/pii/S0306437903001297. doi:https://doi.org/10.1016/j.is.2003.12.001.

[19] G. Moerkotte, T. Neumann, Accelerating queries with group-by and join by groupjoin, Proceedings of the VLDB Endowment 4 (2011) 843–851. doi:10.14778/3402707.3402723.

[20] M. Akinde, M. H. Bhlen, D. Chatziantoniou, J. Gamper, θ-Constrained multi-dimensional aggregation, Information Systems 36 (2011) 341–358. doi:10.1016/j.is.2010.07.005.

[21] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, Dremel: Interactive analysis of web-scale datasets, in: Proc. of the 36th Int'l Conf on Very Large Data Bases, 2010, pp. 330–339. URL: http://www.vldb2010.org/accept.

htm.

[22] A. Rosenthal, et al., Outerjoin simplification and reordering for query optimization, ACM Transactions on Database Systems 22 (1997) 43–74.

6. Acknowledgment

Mohamed Khalefa is partially supported by NSF Grant Award #2131052.