Master Thesis

Efficient Processing of Almost-Homogeneous Semi-Structured Data

Author(s):
Yu, Daniel

Publication Date:
2018

Permanent Link:
https://doi.org/10.3929/ethz-b-000272311

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master’s Thesis Nr. 195
Systems Group, Department of Computer Science, ETH Zurich

Efficient Processing of Almost-Homogeneous Semi-Structured Data

by
Daniel Yu

Supervised by
Prof. Dr. Gustavo Alonso,
Dr. Ingo Müller,
Dr. Ghislain Fourny

October 2017 - March 2018
Abstract

For large scale data processing applications, users have the choice between semi-structured data and relational data. Semi-structured data have the advantage that the schema is not needed upfront. But the flexibility comes at the price of performance. In practice however, semi-structured datasets do not use the full flexibility and are usually to a certain level homogeneous. We call it almost-homogeneous. Such an almost-homogeneous dataset cannot be processed by relational database systems because of the small part of heterogeneity. Systems built for semi-structured data on the other hand sacrifice efficiency for flexibility, even with a large portion of the data being homogeneous.

Instead, we envision a system that can gradually adapt to the homogeneity of the data. The performance degrades gracefully with the degree of heterogeneity. In the best case it can process data at speed of modern relational data systems if the input is fully homogeneous, but it is able to handle heterogeneous data with an overhead cost. This thesis describes the theoretical aspect of such a system.

In this thesis we first cover the theoretical mind-set and framework to characterise the degree of almost-homogeneity in datasets. We then use the framework to show the evidence of almost-homogeneity in common JSON datasets. We propose a novel theoretical approach to bring the semi-structured JSONiq world to the relational by redefining the JSONiq expressions and introducing new relational operators in order to process JSONiq query on relational tables. One of the main contributions are the two mappings: the mapping from JSONiq data model to relations and the mapping from JSONiq expression to relational expressions. The mappings allow us to process semi-structured data with relational operators.
## Contents

1. Introduction 4

2. Background and Related Work 7
   2.1. NoSQL .......................... 7
   2.2. JSON .......................... 8
      2.2.1. Data Types ................. 8
      2.2.2. JSON Lines ................ 9
   2.3. JSONiq ........................ 9
      2.3.1. Data Model ................. 10
      2.3.2. Expressions ............... 10
      2.3.3. Recursive Definition of Expressions ................. 10
      2.3.4. Implementations ........... 12
   2.4. Relational Algebra on Sequences ................. 12
      2.4.1. Pivot/Shred Operator ........ 14
      2.4.2. Special Functions and Operators ............... 16
      2.4.3. Shortcuts ................. 17

3. Data Analysis 19
   3.1. Homogeneity in JSON Lines datasets ................. 19
      3.1.1. Columns in JSON Lines ........ 20
      3.1.2. Reddit ................ 20
      3.1.3. New York Times ........... 21
      3.1.4. Yelp ................ 22
      3.1.5. Urban Dictionary ........... 22
      3.1.6. Conclusion ............... 23
   3.2. Data Split ................ 27
      3.2.1. Schema ............... 27
      3.2.2. Coverage Graph ........... 27
      3.2.3. Reddit ............... 29
      3.2.4. New York Times ........... 30
      3.2.5. Urban Dictionary ........... 32
      3.2.6. Yelp ............... 33
      3.2.7. Conclusion ............... 34

4. Mapping 35
   4.1. Redefinition of Expressions ................. 35
      4.1.1. Sequence of Items as Dynamic Context ................. 35
1. Introduction

Semi-structured data format, like XML and JSON, have the advantage that no schema definition is upfront needed for processing. But the flexibility comes with a cost. The heterogeneity introduces an overhead in processing untyped and schema-less data. On the other hand, relational database systems are fast and well-optimised with decades of researches and optimisations, but it does not support heterogeneous data. Both choices have a trade-off between speed or flexibility.

In practice, we can observe that most collections of data do not use the properties of JSON and XML to its full extents and the level of heterogeneity is limited. With almost-homogeneous we denote datasets that have a large degree of homogeneity. However, such datasets cannot be processed by relational database systems because of the small portion of heterogeneity and semi-structured database systems sacrifice performance for flexibility even with a large portion of the data being homogeneous.

Instead, we envision a system that can gradually adapt to the homogeneity of the data. The goal of this thesis is to explore parts of designing such a system. Depending on the heterogeneity of the data, we propose a system that can switch between a highly optimised relational execution and the slower more flexible execution. The system can adapt to the level of heterogeneity: It can process data at the speed of modern relational database systems if the input data is completely homogeneous, but it is able to handle heterogeneous data with an overhead cost. To build such a system we need to discuss two parts, the data part and the query processing part. Figure 1.1 illustrates the architecture we envision.
The data part covers the data analysis where we sample a dataset to generate the meta data and the split of the data. The idea is to have a split such that one part of the data is homogeneous. The homogeneity helps to optimise the query processing part in a later stage.

The query processing starts with a JSONiq query. In this thesis we assume that the lexer and parser are already given. We start with an expression that represents the JSONiq query as specified on the webpage [13]. The goal is to transform the JSONiq query to relational operators. The mapping itself is done in two steps. First, we map the JSONiq expression tree to relational operators without further information. In a second step we optimise the relational plan by using additional meta information about the data, in particular the data split and the column information of the homogeneous part of the data. The idea for the next step is to produce C++ code from the optimised relational plan and use a JIT compiler to execute the mapped JSONiq query on the mapped data. For the scope of this thesis, we concentrate on the JSONiq mapping and the optimisation with the data split information.

In Chapter 2 we discuss the background about JSONiq and relational algebra. Chapter 3 covers the definition and framework to measure the homogeneity in datasets. We use the framework to analyse four datasets to find evidence of almost-homogeneity in practise.
The Chapter 4 defines formally two mappings: 1. The mapping from JSONiq data model to relations and 2. The mapping from JSONiq expression to relational expressions. We then briefly cover in Chapter 5 the optimisation of the mapped relational algebra to an efficient query plan.
2. Background and Related Work

2.1. NoSQL

With the rise of internet applications during the late 90s and early 2000s, the basic requirements for the database systems shifted to a design that can offer the flexibility and dynamic scaling to serve millions of users on the public internet. By the late 2000s, various new non-relational database technologies had emerged. These technologies can be described as NoSQL (also known as "not only SQL"). The most famous examples are: document databases, graph stores, key-value stores and wide-column stores [8].

In this thesis we concentrate on the document databases, more specifically on JSON document databases.

The NoSQL technologies also introduce some changes to the data model concept and the schema concept.

**Data model**: SQL databases are extremely efficient at storing structured data, but as unstructured data has grown rapidly (e.g. social media posts), workarounds and compromises are needed. NoSQL supports changing data types: structured, semi-structured, unstructured data [10] [9].

In JSON document databases the data model is semi-structured. A main property of the semi-structured data model that differs from the relational data model is the nestedness. In JSON the data model is described recursively. It allows to nest objects and arrays freely with as many levels as desired. For example, the following JSON describes a JSON object with an array with an object in an object.

```json
{
    "nest1": [
        {
            "nest2": {
                "nest3": true
            }
        }
    ]
}
```

**Schema**: In document databases there is no need for a fixed schema upfront. Without a schema, a type guarantee is not given. Two JSON objects in the same collection could have different types for the same key. For example, the two representations of users in the same collection are valid.
2. Background and Related Work

Note that in one object the age is an integer, whereas the second one is of type string. JSON document stores allow the flexibility not to specify a schema upfront for the data it stores.

2.2. JSON

As stated on json.org, JSON is a "lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate." [1]

2.2.1. Data Types

The data model behind JSON is built upon the concept of items. Items are the most generic data type in JSON. There are two categories in JSON data types: primitive and structured. Primitive types are atomic types, that cannot be split into smaller parts. On the other hand, structured items are made up of items. These items can be atomic items or even structured items (nesting). Figure 2.1 illustrates the hierarchy of JSON data types.

**object** Object are one of the two structured items in JSON. An object can be described as an unordered collection of key-value pairs, where the key is of type string. We do not allow duplicate keys.

**array** Arrays are the second structured item in JSON. An array is an ordered list of items.

**string** Strings are one of the four primitive items in JSON. A string is a sequence of zero or more Unicode characters.

**number** Numbers are also one of the four primitive items in JSON. They can be represented in the following three ways: integers (positive and negative natural numbers), fractions (with a dot) and exponent format (the scientific notation with an e).
null Null values are the next primitive items in JSON. A null value represents an empty value.\footnote{Please note that the null value and absence of a value are not treated in the same way in JSON. The same applies to the null value and an empty object/array.}

boolean Booleans are the last primitive items in JSON. A boolean can have one of two values: true or false.

![Hierarchical JSON data types](image)

Figure 2.1.: Hierarchical JSON data types.

### 2.2.2. JSON Lines

JSON Lines is a text format description to store multiple JSON items. For further information see [2]. The format consists of three simple rules:

**UTF-8 Encoding** The encoding of the data is required to be UTF-8.

**JSON Line** Each line is a valid JSON item. "The most common values will be objects or arrays, but any JSON value is permitted." [2]

**Line Separator** The line separator is required to be '\n'

In this thesis we use a collection of JSON objects as input data. As input data format, we use JSON Lines and treat each line as one record in the collection.

### 2.3. JSONiq

JSONiq is a functional and declarative language for processing and querying nested, heterogeneous, and semi-structured JSON data developed by Jonathan Robie, Ghislain Fourny, Matthias Brantner, Daniela Florescu, Till Westmann, Markos Zaharioudakis [3].
2. Background and Related Work

2.3.1. Data Model

The main data model for JSONiq are the concepts of sequence of items and dynamic context.

sequence of items As the name suggests, a sequence of items is a sequence of zero or more JSON items. We use comma separated format to visualize a sequence. Consider the following example.

("foo", 2, true, { "foo": "bar" }, null, [1, 2, 3])

The example represents a sequence of six JSON items: A string, a number, a boolean, an object with one key-value pair, a null and an array with three numbers.

dynamic context A dynamic context is a set of key-value pairs, where the key is a variable name starting with a dollar sign and the value is a sequence of items. The following example represents a dynamic context.

< $i : (1, 2, true, false), $j : (), $k : ("bla") >

The example shows a dynamic context with three variables: $i$, $j$ and $k$. $i$ is a sequence of four JSON values, $j$ an empty sequence and $k$ a single string. In JSONiq the official name is tuples. To avoid ambiguity with relational tuples we renamed it to dynamic context.²

2.3.2. Expressions

In JSONiq, expressions are the main building blocks for querying, modifying, and creating semi-structured data. Formally, it takes a dynamic context $DC$ and returns a sequence of items $S$.

$$ e \in E : DC \rightarrow S $$

2.3.3. Recursive Definition of Expressions

In JSONiq expressions can be defined recursively. As shown in Figure 2.2 the recursively defined expression can include multiple subexpressions $e_1, \ldots, e_n$.

²The dynamic context is already defined in JSONiq. It describes a large construct that also includes the JSONiq tuples along with other information. In our case when we use the word dynamic context, we mean the JSONiq tuples.
2. Background and Related Work

Each of the subexpressions $e_j$ gets as an input $D_{\text{in}}^j$, which is defined by a function $\delta_j$. $\delta_j$ can take all the past outputs $S_{\text{out}}^1, \ldots, S_{\text{out}}^{j-1}$ and the main input $D_{\text{input}}$ as argument. The functions $\delta_1, \ldots, \delta_n$ are defined by the expression type. Note that the subexpressions have a specific order in the expression itself.

Let us take the example expression $i + 2 \leq 5$. The illustration of the example expression is shown in Figure 2.3.

To evaluate the expression, we need a dynamic context. Let us use the following simple dynamic context $D_{\text{in}}$ as input.

$< i : (5) >$

In our example dynamic context, the number 5 is bound to the variable $i$. 

---

Figure 2.2.: Illustration of a recursively defined expression.

Figure 2.3.: Illustration of the example expression.
2. Background and Related Work

We start at the top of the expression tree, the comparison expression $e_1$. The comparison expression $e_1$ takes $DC_{input}$ and redirects it without transformation to the subexpressions $e_2$ and $e_3$. Note that the expression $e_1$ treats the two subexpressions as black boxes. The view of expression $e_1$ is illustrated below in Figure 2.4.

![Figure 2.4: Illustration of the black box view for the outer comparison expression.](image)

Subexpression $e_2$ is a basic arithmetic expression and redirects its input without modification to the subexpressions $e_3$ and $e_4$. Subexpression $e_3$ is a variable reference, that looks up in his input dynamic context the reference $i$ and returns the value. In our case it is 5. The next subexpression is $e_4$, which is an atomic literal. Independently of the input, $e_4$ always returns the value 2. With both results of subexpressions $e_3$ and $e_4$, the basic arithmetic expression $e_2$ adds them together and returns the result. In our example $e_2$ returns 7. The subexpression in $e_5$ is an atomic literal, that returns the number 5 independently of the input. The comparison expression $e_1$ finally takes both return values, 7 and 5, of the two subexpressions and compares them. As 7 is not less or equal to 5, the most outer expression $e_1$ returns the sequence of a single item: false.

2.3.4. Implementations

There are several implementations of JSON databases with JSONiq query language support, the most notable ones are Zorba and Apache VXQuery.

2.4. Relational Algebra on Sequences

In JSONiq one of the main data model concept is the sequence of items, which has an ordering. For example, the two sequences

$$(1, 2, 3) \text{ and } (3, 2, 1)$$

are two different sequences as the ordering matters.
The traditional relational algebra introduced by Codd [5] is defined on sets. Sets do not have a notion of tuple ordering, however we need the ordering as the basic data model in JSONiq is sequences. There are two ways to introduce an ordering in relational algebra.

**Add an ordinal attribute** The first option would be to add in each table an ordinal attribute that keeps track of the ordering. With this option we do not have to change the relational algebra. The two tables below are equivalent, but with the ordinal value the ordering is given.

<table>
<thead>
<tr>
<th>id</th>
<th>ordinal</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>1</td>
<td>&quot;Banana&quot;</td>
</tr>
<tr>
<td>id2</td>
<td>2</td>
<td>&quot;Apple&quot;</td>
</tr>
<tr>
<td>id3</td>
<td>3</td>
<td>&quot;Orange&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>ordinal</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>1</td>
<td>&quot;Banana&quot;</td>
</tr>
<tr>
<td>id3</td>
<td>3</td>
<td>&quot;Orange&quot;</td>
</tr>
<tr>
<td>id2</td>
<td>2</td>
<td>&quot;Apple&quot;</td>
</tr>
</tbody>
</table>

**Relational algebra on sequences** The second option would be to redefine all the relational operators such that the operators support the operation on tuples with an ordering. While there are studies about relational algebra for sequences [7] [14], they do not directly fit to the processing nature of JSONiq. JSONiq operates on dynamic context that introduces another level of nestedness.

In this thesis we will use the latter approach. A sequence is a data structure where the items in the sequence have an ordering. In this thesis, we define a relation to be a sequence of tuples instead of sets which introduces an ordering of the tuples. For the operator we use in the thesis, we define the output ordering in the following way:

**Projection** A projection operator \( \pi \) selects a subset of the columns. We define that the projection operator on sequences maintains the order of the tuples after the operation.

**Selection** A selection operator \( \sigma \) selects a subset of the tuples of the relation. A predicate function defines which tuples are selected. We define that the selection operator on sequences maintains the order of the tuples after the operation.

**Rename** A rename operator \( \rho \) renames an attribute name. We define that the rename operator maintains the order of the tuples after the operator.

**Cartesian product** A cartesian product operator \( \times \) combines the tuples of one relation with all the tuples of the other relation. We define that the order of the sequence is given by its lexicographical order.

**Stable sort** A stable sort operator \( \tau \) sorts the tuples by one or more attributes. We define that the order is given first by the sorting key and second by the input ordering of the tuples.

**Join** A theta join operator \( \bowtie \) is a combination of the cartesian product that keeps the sequences in lexicographical order, followed by a selection, that also maintains the order of the tuples.

13
2.4.1. Pivot/Shred Operator

In JSONiq it is common to transform column names to values and vice versa. As the relational algebra does not provide such a functionality, we would like to introduce two new operators, the pivot and the shred operator.

**Pivot operator**

The pivot operator takes three parameters, id, column and value. It works like the pivot function from Microsoft Excel. We denote the pivot function with

\[
pivot_{id,\text{column},\text{value}}(R_{\text{input}})
\]

Suppose we have the following input table \( R_{\text{input}} \).

\[
R_{\text{input}} = \begin{array}{ccc}
\vdots & \vdots & \vdots \\
(id) & (col) & (val) \\
\vdots & \vdots & \vdots \\
\end{array}
\]

The pivot operator first groups the tuples by id. For each group key \( id_i \) we have possibly multiple tuples belonging to the same group key. The following table illustrates a table of tuples belonging to the same group key \( id_i \).

\[
\begin{array}{ccc}
id & \text{column} & \text{value} \\
id_i & col_1 & val_1 \\
id_i & col_2 & val_2 \\
\vdots & \vdots & \vdots \\
id_i & col_m & val_m \\
\end{array}
\]

For the tuples belonging to group key \( id_i \) we build a new output tuple.

\[
tuple_i = \begin{pmatrix}
(id \rightarrow id_i) \\
(col_i \rightarrow val_i) \\
\vdots \\
(col_m \rightarrow val_m)
\end{pmatrix}
\]

Note that the columns \( col_1, \ldots, col_n \) are unique. Otherwise the operator throws an error. We repeat the process for all the group key \( id_i \) and concatenate the resulting tuples.\(^3\)

The result is our output relation.

\[
R_{\text{output}} = \Gamma_{i=1}^{m} \tuple_i
\]

Let us illustrate the pivot operator with a small example. Consider the following input table.

\(^3\)We denote with \( \Gamma \) the concatenation of items. For definition refer to Subsection 2.4.2
Using our pivot operator, the following output table is produced.

\[
R_{\text{output}} := \text{pivot}_{cid,\text{column},\text{value}}(R_{\text{input}}) =
\]

<table>
<thead>
<tr>
<th>cid</th>
<th>column</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;A&quot;</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>&quot;B&quot;</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>&quot;C&quot;</td>
<td>false</td>
</tr>
<tr>
<td>3</td>
<td>&quot;B&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>

Shred operator

The shred operator is the inverse operator of the pivot operator. Same as the pivot operator, the shred operator takes three parameters.

\[
\text{shred}_{id,\text{column},\text{value}}(R_{\text{input}})
\]

Suppose we have the following input table \(R_{\text{input}}\).

\[
\begin{array}{cccc}
\text{id} & \text{col}_1 & \text{col}_2 & \cdots & \text{col}_m \\
\hline
\text{id}_1 & \text{val}_{1,1} & \text{val}_{1,2} & \cdots & \text{val}_{1,m} \\
\text{id}_2 & \text{val}_{2,1} & \text{val}_{2,2} & \cdots & \text{val}_{2,m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\text{id}_n & \text{val}_{n,1} & \text{val}_{n,2} & \cdots & \text{val}_{n,m} \\
\end{array}
\]

The operator goes through all the values \(\text{val}_{i,j}\). If the value is not absent \(\bot\), we produce a new tuple with the current column name as value for the key \text{column} and the value \(\text{val}_{i,j}\) as value for the key \text{value}.

\[
\Gamma_{i=1}^{n} \Gamma_{j=1}^{m} \left( \begin{array}{ccc}
\text{id} & \rightarrow & \text{id}_i \\
\text{column} & \rightarrow & \text{col}_j \\
\text{value} & \rightarrow & \text{val}_{i,j} \end{array} \right)
\]

Let us illustrate the shred operator with a small example. Consider the following input table.

\[
R_{\text{input}} = \begin{array}{cccc}
\text{id} & \text{A} & \text{B} & \text{C} \\
\hline
1 & 1 & 2 & \bot \\
2 & \bot & 3 & 4 \\
\end{array}
\]

Using our shred operator, the following output table is produced.
\[ R_{output} = shred_{cid,colmn,value} (R_{input}) = \]

<table>
<thead>
<tr>
<th>cid</th>
<th>column</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;A&quot;</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>&quot;B&quot;</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>&quot;B&quot;</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>&quot;C&quot;</td>
<td>4</td>
</tr>
</tbody>
</table>

An interesting observation is the fact that projection is no longer a primitive operator. Projection can be described as a special case of shredding followed by a selection and a pivot operator.

2.4.2. Special Functions and Operators

Distinct Operator

In a sequence the same tuple can appear multiple times at multiple positions. The distinct operator scans the sequence and removes all duplicate tuples it has already encountered before. The output sequence has the same ordering as before.

To illustrate the operator, consider the following input relation with three tuples where two tuples are identical.

\[
R_{input} = \begin{array}{|c|c|}
\hline
col1 & col2 \\
1 & "A" \\
1 & "A" \\
2 & "A" \\
\hline
\end{array}
\]

Using our distinct operator, it returns a relation with two tuples.

\[
R_{output} = \text{distinct} (R_{input}) = \begin{array}{|c|c|}
\hline
col1 & col2 \\
1 & "A" \\
\hline
\end{array}
\]

Concatenation operator

The concatenation math operator takes multiple items or sequences and produces a new sequence where all the items are concatenated. Please note that sequences cannot be nested, therefore a concatenation of sequences produces a new sequence with all the items concatenated. We denote the concatenation operator with \( \Gamma \).

\[
\Gamma_{i=1}^{n} x_i = (x_1, x_2, \ldots, x_n)
\]

For example, we have the following three items, a sequence and two integers.

\[
x_1 = 1, \quad x_2 = (2, 3, 4), \quad x_3 = 5
\]

The concatenation returns the following result.

\[
\Gamma_{i=1}^{3} x_i = (1, 2, 3, 4, 5)
\]
id() Function

The id() function is used in this thesis to generate new identifiers. To keep it simple it returns integers from 1 onwards. To illustrate this function, we use the following input table.

\[
R_{input} = \begin{array}{|c|c|}
\hline
\text{col1} & \text{col2} \\
\hline
A & a \\
B & b \\
C & c \\
\hline
\end{array}
\]

The following projection creates a column cid where the entries starts at 1 and increases for each entry by one.

\[
R_{output} := \pi_{cid \leftarrow \text{id}()}, \text{col1}, \text{col2} (R_{input}) = \begin{array}{|c|c|c|}
\hline
\text{cid} & \text{col1} & \text{col2} \\
\hline
1 & A & a \\
2 & B & b \\
3 & C & c \\
\hline
\end{array}
\]

ref_id() Function

The ref_id() function works in the same way as the id() function. In this thesis we use the ref_id() function to generate new reference identifier. To have a visual difference we start each reference identifier with a "ref_". To illustrate the function, we use the same example table \( R_{input} \) as above.

\[
R_{output} := \pi_{ref \leftarrow \text{ref_id}()}, \text{col1}, \text{col2} (R_{input}) = \begin{array}{|c|c|c|}
\hline
\text{ref} & \text{col1} & \text{col2} \\
\hline
\text{ref}_1 & A & a \\
\text{ref}_2 & B & b \\
\text{ref}_3 & C & c \\
\hline
\end{array}
\]

2.4.3. Shortcuts

Join by cid

We denote with \( R_1 \bowtie_{cid} R_2 \) the natural join on the attribute cid. If there are other common attributes, we rename them by prepending the table name to the common attributes. Consider the following two tables.

\[
R_1 = \begin{array}{|c|c|c|}
\hline
\text{cid} & \text{col1} & \text{col2} \\
\hline
1 & A & x \\
2 & B & y \\
\hline
\end{array} \quad R_2 = \begin{array}{|c|c|}
\hline
\text{cid} & \text{col1} \\
\hline
1 & a \\
2 & a \\
2 & b \\
\hline
\end{array}
\]

Our join returns the following output table.
2. Background and Related Work

\[ R_{\text{output}} := R_1 \bowtie_{\text{cid}} R_2 = \begin{array}{cccc}
\text{cid} & R_1.\text{col1} & R_2.\text{col1} & \text{col2} \\
1 & A & a & x \\
2 & B & b & y \\
2 & B & b & y \\
\end{array} \]

Note that \text{col1} has been renamed to \(R_1.\text{col1}\) and \(R_2.\text{col1}\) as it is a common attribute in both tables that is not the join attribute.

**Remove Columns**

To remove columns, we use the following notation.

\[ \pi_{*\setminus\{\text{col}_1,\ldots,\text{col}_n\}}(R) \]

This projection selects every column where the column name is not in \(\{\text{col}_1,\ldots,\text{col}_n\}\). We use this notation as a shortcut for

\[ \text{pivot}_{\text{cid},\text{col},\text{val}}(\sigma_{\text{col} \neq \text{col}_1 \text{ and } \ldots \text{ and } \text{col} \neq \text{col}_n}(\text{shred}_{\text{cid},\text{col},\text{val}}(R))) \]

**Projection by Variable Name**

For our mappings, it is very common to project by a specific variable name. We denote the operation as follows.

\[ \pi_{\$\text{varname}^*}(R) \]

This projection selects every column which is \$\text{varname}\ or starts with \$\text{varname}. (note the dot). It is a shortcut for

\[ \text{pivot}_{\text{cid},\text{col},\text{val}}(\sigma_{\text{col} = \$\text{varname} \text{ or col.startsWith("$\text{varname.")}}}(\text{shred}_{\text{cid},\text{col},\text{val}}(R))) \]

**Rename a Variable Name**

With the following shortcut, we denote the rename of a variable name, which is used in our mappings.

\[ \rho_{\$\text{oldname}\rightarrow\$\text{newname}^*}(R) \]

For every column which is \$\text{oldname}\ or starts with \$\text{oldname}. (note the dot), we replace the \$\text{oldname}\ with \$\text{newname}.

Consider the following table as our input example.

\[
R_{\text{input}} = \begin{array}{cccc}
\text{cid} & \$i & \$i.\text{name} & \$i.\text{age} & \$\text{in} \\
1 & \perp & "A" & 20 & \text{true} \\
2 & \perp & "B" & 25 & \text{false} \\
\end{array}
\]

The relational expression \(\rho_{\$\text{i} \rightarrow \$\text{new}}(R_{\text{input}})\) outputs the following table \(R_{\text{output}}\).

\[
R_{\text{output}} = \begin{array}{cccc}
\text{cid} & \$\text{new} & \$\text{new.\text{name}} & \$\text{new.\text{age}} & \$\text{in} \\
1 & \perp & "A" & 20 & \text{true} \\
2 & \perp & "B" & 25 & \text{false} \\
\end{array}
\]
3. Data Analysis

One of the reasons why a JSON document is slower in processing than conventional relational database is the lack of a closed schema. Such a schema is not mandatory in document stores for the sake of flexibility. The lack of a schema can hit the performance in two ways:

1. The overhead introduced by type checking and
2. The lack of type information prevents optimisation on the query plan.

However, in reality the datasets do not use the flexibility of JSON document stores to its full extent. There are cases where the data is even fully relational in the sense that the schema of the data is fully homogeneous. In such cases, the slowdown of the processing in document store is not justified.

Algorithms for schema inference for a whole JSON dataset have been shown in several studies [6][4]. Instead of inferring the schema from the data, our approach is to let the user choose the openness of a schema, with which we split the data in two parts. The homogeneous part is compliant to the schema that gives us the type information for further optimisation. We leave the heterogeneous part without a schema. With our approach the user can choose the trade-off between openness of the schema and the size of the homogeneous part in a finer granularity that indirectly also affects the performance of the system. To give a guidance to split, we introduce a novel visualization of the dataset.

In this chapter we show evidences that in practice the datasets do not use JSON properties like nestedness and heterogeneity to its full extent. With the evidences we then argue, that the splitting in a homogeneous and heterogeneous part is reasonable in practice. For this purpose, we look at four well-known datasets from various sources: Reddit, New York Times, Urban Dictionary and Yelp. The sources of the datasets are listed in the appendix section.

3.1. Homogeneity in JSON Lines datasets

To support our approach of splitting the data into two parts, the homogeneous and the heterogeneous part, we show with four datasets, that the flexibility of JSON documents are used parsimoniously in practise. Especially the variety of types of a column. Most of the datasets only use one or two types for a column. We first define formally columns in JSON Lines and then analyse the type distribution of the columns.

---

1 The sources of the datasets are listed in the appendix section.
3. Data Analysis

3.1.1. Columns in JSON Lines

We assume, that each line in our JSON Lines dataset is a JSON object. We define the column of a JSON Lines dataset to be the concatenation of a specific field keyname of all the objects in a given dataset. In case of absence of the keyname, we take the special character ⊥ to mark the absence. The output is a sequence of the items.

\[
\text{column(keyname, dataset)} = \Gamma_{o \in \text{dataset}} \begin{cases} 
  o.\text{keyname}, & \text{if } o \text{ contains keyname} \\
  \bot, & \text{otherwise.}
\end{cases}
\]

The type distribution of a column is the type distribution of the values. In our analysis, we normalized the type distribution to the number of non-absence values. In addition to the type distribution we use the term exist percentage to measure the percentage of values which are not absent.

Example dataset:

```json
{"name": "Tom", "age": 25}
{"name": "Peter", "age": "20"}
{"name": "Sarah"}
```

The example dataset above has two columns, name and age. To calculate the distribution of the types we first analyse the exist percentage. For the column name the exist percentage is 100% as all the objects in the dataset has the field name. The type distribution of it has a 100% type string and 0% for every other type. The column age has an exist percentage of 66.6%, as one out of three objects does not have the field age. The type distribution, normalized to the number of non-absence values, is 50% string and 50% integer.

With the type distribution definition, the next subsections describe the analysis of four dataset to show that in practice, the distribution is often limited to only one or two types.

3.1.2. Reddit

Reddit is a community-based web platform for sharing, voting and commenting ideas [12]. The dataset consists of Reddit comments from the years 2006 to 2016. The total data size is 1.6 terabytes. Each line in the dataset is a comment represented by an JSON object. The JSON object is a flat object with up to 34 fields. A sample JSON object:

```json
{
  "subreddit": "reddit.com",
  "created_utc": 1136074029,
  "score": 0,
  ... 
  "body": "early 2006 a probable date",
  "author": "jh99",
  "parent_id": "t3_22569",
}
```
To bring the data to a reasonable size for the analysis, a subset of the dataset has been generated by taking every 1000th line.

The type distribution analysis is shown in Table 3.1 on page 24. The Reddit dataset has in total 34 columns. About half of them do not contain any absent value. For the type distribution, several patterns can be observed. 29 out of 34 columns just use one type. Three columns use a combination of string and null without any absence value and two columns use more than two types.

We can conclude that except for two columns, create_utc and edited, the columns only use one or two types. In case of two types, it is used with an atomic type along with the null type.

3.1.3. New York Times

The New York Times is an American newspaper, which is distributed digitally and in print [11]. The dataset consists of news articles from 1900 to 2016. The total size is around 15 gigabytes. Each line represents a New York Times article as a JSON object. Some of them nest other objects or array of objects. An example article JSON object:

```
{
    "web_url": "https://www.nytimes.com/2016/12/30/us/
cub-scouts-transgender.html",
    "snippet": "Three years after the Boy Scouts ended its ban...",
    ...
    "multimedia": [
        {
            "width": 75,
            "type": "image"
            ...
        },
        {
            "width": 600,
            "type": "image"
            ...
        },
    ],
    ...
}
```

The type distribution analysis is shown in Table 3.2 on page 25. The New York Times dataset has in total 20 columns. None of them contains absent values. For the type distribution, we have eleven columns that only use one type. Eight columns use two
types in a specific pattern that consist of one type and the null type. Only one column, \textit{word\_count}, uses two types, where none of them is the null type.

As in the Reddit dataset, we can conclude, that the type distribution of the columns mainly has two patterns, only one type or one type with null support.

3.1.4. Yelp

The purpose of Yelp is "to connect people with great local businesses" [15]. The main function of Yelp is to list businesses and let customer review them. The Yelp datasets consists of over 150'000 businesses and has a total size of 128 megabytes. Each line in the dataset represents a business listing on Yelp in the JSON format. An example business listing JSON object:

```json
{
    "business_id": "YDf95gJZaq05wvo7hTQbbQ",
    "name": "Richmond Town Square",
    "address": "691 Richmond Rd",
    "postal_code": "44143",
    "stars": 2.0,
    ...
    "categories": ["Shopping", "Shopping Centers"],
    "hours": {
        "Monday": "10:00-21:00",
        "Tuesday": "10:00-21:00",
        "Friday": "10:00-21:00",
        "Wednesday": "10:00-21:00",
        "Thursday": "10:00-21:00",
        "Sunday": "11:00-18:00",
        "Saturday": "10:00-21:00"
    }
}
```

The type distribution analysis is shown in Table 3.3 on page 26. The Yelp dataset has in total 15 columns. We can observe that none of the columns has an absent value. Except two columns, \textit{latitude} and \textit{longitude}, all the columns are only using one type. The other two columns are using the pattern with one type and the null type. However, the percentage of null is extraordinary low.

We can conclude that Yelp is in mostly using one type per column.

3.1.5. Urban Dictionary

Urban Dictionary is an online collaborative dictionary focusing on defining slang words. The dataset consists of 2.6 Million defined words and has a total size of 1.8 gigabytes. Each line represents a word definition in the JSON format. An example JSON object:
3. Data Analysis

The type distribution analysis for Urban Dictionary is shown in Table 3.4 on page 26. The Urban Dictionary dataset has in total 13 columns. We can observe that all the columns just have one type without any absent value. The null type is not used at all.

We can conclude that Urban Dictionary is a fully homogeneous dataset.

3.1.6. Conclusion

With the type distribution of four dataset from well-known websites, we could observe that most of the columns are using two main patterns: 1. One type only or 2. One type with null support.

The second observation is, that the absence value is not used very often. This can be explained with the null usage. Null and absent value are treated differently in JSON. However, in practise using one of them is enough to mark an absent value. In the four datasets, only the Reddit dataset takes the advantage of both, the null and absent value. Reddit is using absent value for optional values. The null value is used where the presence of a field is the information.
<table>
<thead>
<tr>
<th>Column</th>
<th>exist percentage</th>
<th>bool</th>
<th>float</th>
<th>int</th>
<th>null</th>
<th>string</th>
<th>object</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>distinguished</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>99.3871</td>
<td>0.61287</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>retrieved_on</td>
<td>99.79570991</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>likes</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>name</td>
<td>61.89989785</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>link_id</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>saved</td>
<td>0.034048349</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>author_flair_text</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>71.8079</td>
<td>28.1920</td>
<td>0</td>
</tr>
<tr>
<td>created_utc</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>30.1668</td>
<td>0</td>
<td>69.8331</td>
<td>0</td>
</tr>
<tr>
<td>subreddit</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>stickied</td>
<td>30.16683691</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>approved_by</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>body</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>removal_reason</td>
<td>22.54000681</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>score_hidden</td>
<td>61.89989785</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>author_flair_css_class</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>71.8079</td>
<td>28.1920</td>
<td>0</td>
</tr>
<tr>
<td>archived</td>
<td>61.89989785</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>author</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>score</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>parent_id</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>num_reports</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>report_reasons</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>user_reports</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>mod_reports</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>controversy</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>body_html</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ups</td>
<td>92.61150834</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>edited</td>
<td>100</td>
<td>97.3782</td>
<td>0.20429</td>
<td>2.41743</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>id</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>replies</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>downs</td>
<td>61.89989785</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>banned_by</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subreddit_id</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gilded</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>created</td>
<td>0.034048349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.1.: Type distribution of the columns for the Reddit dataset. Numbers in percentage.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>exist</th>
<th>percentage</th>
<th>bool</th>
<th>float</th>
<th>int</th>
<th>null</th>
<th>string</th>
<th>object</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>headline</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>byline</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>42.7480916</td>
<td>0</td>
</tr>
<tr>
<td>web_url</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22.13740458</td>
<td>77.86259542</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>snippet</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>96.18320611</td>
<td>3.816793893</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>subsection_name</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>document_type</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pub_date</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>news_desk</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25.19083969</td>
<td>0</td>
</tr>
<tr>
<td>keywords</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>blog</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>lead_paragraph</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>43.51145038</td>
<td>56.48854962</td>
</tr>
<tr>
<td>source</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>print_page</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7.633587786</td>
<td>92.36641221</td>
</tr>
<tr>
<td>word_count</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9.923664122</td>
<td>0</td>
</tr>
<tr>
<td>slideshow_credits</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>multimedia</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>section_name</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>70.99236641</td>
<td>29.00763359</td>
</tr>
<tr>
<td>abstract</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>type_of_material</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.2.: Type distribution of the columns for the New York Times dataset. Numbers in percentage.
### 3. Data Analysis

<table>
<thead>
<tr>
<th>Column</th>
<th>exist percentage</th>
<th>bool</th>
<th>float</th>
<th>int</th>
<th>null</th>
<th>string</th>
<th>object</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>hours</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>stars</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>attributes</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>review_count</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>neighborhood</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>state</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>postal_code</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>latitude</td>
<td>100</td>
<td>0</td>
<td>99.99936</td>
<td>0</td>
<td>0.000638</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>city</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>name</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>is_open</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>longitude</td>
<td>100</td>
<td>0</td>
<td>99.99936</td>
<td>0</td>
<td>0.000638</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>address</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>business_id</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>categories</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

**Table 3.3.** Type distribution of the columns for the Yelp dataset. Numbers in percentage.

<table>
<thead>
<tr>
<th>Column</th>
<th>exist_percentage</th>
<th>bool</th>
<th>float</th>
<th>int</th>
<th>null</th>
<th>string</th>
<th>object</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>sounds</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>thumbs_up</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>thumbs_down</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>example</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tags</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>lowercase_word</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>definition</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>defid</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>permalink</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>word</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>current_vote</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>author</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>_id</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 3.4.** Type distribution of the columns for the Urban Dictionary dataset. Numbers in percentage.
3. Data Analysis

3.2. Data Split

From the type distribution analysis in Section 3.1 we have observed two patterns, that are used in columns, a single type or a single type with null support. We also observed that sometimes values can be absent. These are valuable information to define our schema.

Recall that our approach is to split our data for processing into two parts, the homogeneous and the heterogeneous part. The splitting is dependent on the chosen schema. With the pattern we learned, we define the schema in the following Subsection 3.2.1. We then introduce the coverage graph, that illustrates the dependency between the chosen schema and the split. The coverage analysis for four datasets is then discussed in the Subsections 3.2.3 to 3.2.6 where we show that our defined schema produces a reasonable data split.

3.2.1. Schema

From the type distribution analysis in Section 3.1 we have observed that most columns only have one type or one type with null support. For a closed schema, restricting to one type is preferred as it allows us to optimise the query plan and avoid type check overheads during processing. For this reason, we propose four different column constraints, all of them restrict the column to only one type with various other supports that are common for JSON Lines datasets.

- **type** This constrains the column to only have one primitive type, such as integer or string. We also require the column to have no missing value at all.
- **type|null** This constrains the column to have one type or to be null. Having one type and null is a very common pattern in JSON.
- **type?** This constrains the column to have one type, but the value can be absent.
- **(type|null)?** This constrains the column to have one type or to be null. It supports missing values as well.

We define a schema to be a set of column constraints. We say an object is following the schema if each of the column constraints in the schema is fulfilled by the object.

3.2.2. Coverage Graph

As stated before the split of the data depends on the chosen schema. To help the user choose and to get further insight of the dataset, we would like to visualize the trade-off between the closedness of the schema and the split. First, we need to define how we measure the closedness of a schema and the split.

**Schema closedness** We measure the closedness of a schema in percentage of columns, that are constrained with one of the four constraints we have introduced earlier, relative to the number of total columns of the dataset. We want to maximize the
closedness of the schema as it gives us more type information of the homogeneous part of the data.

**Schema compliance** We measure the split by the size of the homogeneous part that is the number of objects in the dataset, that is compliant to the schema. The homogeneous part is the one we would like to maximize as it allows the part of the data, that can be optimised, to be bigger. We call the percentage of the size of the homogeneous part item coverage.

To measure the trade-off and to generate the graph, a greedy method is chosen. We first sort the columns by their percentage of the most used type in a decreasing order. Exactly in this order we choose the columns, which need to be constrained, incrementally. For each new chosen column, we then measure the item coverage, to be specific the percentage of items which still follows the constraints of all previous columns. We repeat the same procedure for all constraints. The measured point is then illustrated in a 2D graph. A point \((x, y)\) in the coverage graph means that with \(x\%\) of the columns constrained we have a schema compliance (or item coverage) of \(y\%). The coverage graph for the datasets are explained in the next subsections.
3.2.3. Reddit

The coverage graph in Figure 3.1 shows that the two constraints, type and type/null, do not fit well with the dataset. Even by constraining just one column, the item coverage immediately drops down to zero percent. The type? constraints have full item coverage upon 30% of the columns, it then drops to around 80% item coverage until 85% of the columns constrained. The (type/null)? which supports both, null and missing value, has full coverage for up to 85% of the columns constrained. Even with all the columns constrained we have an item coverage of 70%.

The analysis shows that for an acceptable item coverage, we need at least missing value support. The null support alone does not make a difference of the coverage, but the support for null and missing value gives us the best item coverage.
3.2.4. New York Times

The coverage graph in Figure 3.2 shows us that all the three constraints, type, type/null and type?, perform rather poorly. By constraining more than 20-30% of the columns, we lose almost all the item coverage. By supporting both, the null and missing values, with (type/null)? we can get full coverage while having up to 70% column constrained. With all the columns constraints, we still get an 90% item coverage.

For the nested objects, an analysis has been done separately. Below in Figure 3.3 is the coverage plot for one nested object, that belongs to the key "keywords". For the other five nested objects, the graph looks similar.
Figure 3.3.: Nested Coverage Analysis of the NYT dataset for the objects in the keywords array. The blue line is behind the red line and the green line is behind the purple line.

In the graph from Figure 3.3 we can observe that we get full coverage with type? or (type|null)?. We can conclude that for the nested objects the missing value support is enough.

Considering both analyses, we can conclude that New York Time is using one of the JSON property extensively, nesting. Six columns out of 22 fields contains nested JSON objects. For homogeneity, 90% of the items of the most outer JSON objects are fully homogeneous by using null and absence supports. We can conclude that the New York Times dataset is almost-homogeneous where only 10% of the data is heterogeneous.
3.2.5. Urban Dictionary

![Coverage plot for UD:root](image)

Figure 3.4.: Coverage Analysis of the Urban Dictionary dataset. The blue line is behind the red line and the green line is behind the purple line.

From the graph in Figure 3.4 we can see that the two constraints, *type* and *type/null*, perform badly. With a 40% column constraint the item coverage drops to 10%. Interestingly we get a full item coverage with the other two constraints, *type?* and *(type/null)?*, even with 100% of the columns constraints.

We can conclude that Urban Dictionary is not using the two JSON properties, nestedness and heterogeneity, much. Except one column, there is no nested object in the word definition JSON object. By supporting missing values, all the columns can be constrained to one type for all the items.

Urban Dictionary is most likely using a simple homogeneous database. JSON happens to be the choice for the format of transport.
3.2.6. Yelp

From the graph in Figure 3.5 we can see that for all four constraints, \textit{item}, \textit{item?}, \textit{item/null} and \textit{(item/null)?}, we get almost full (>99.99\%) item coverage with all the columns constrained. For the nested objects, we do for each field a separate analysis. The following coverage graph is for the hours field. All the other 9 fields are looking similar.
Figure 3.6.: Nested Coverage Analysis of the Yelp dataset for the hour field. The green line is behind the purple line and the blue line is behind the red line.

From the graph in Figure 3.6 we can observe that two constraints, type? and (type/null)?, are performing very well. They show a full item coverage with all the columns constrained. The other two constraints, type and (type/null), give a suboptimal solution.

We conclude that the absence support is enough to get a full coverage for business object and all its nested objects. Yelp is only using the nestedness property of JSON. The heterogeneity property is not used much by Yelp. All the columns can be constraints to one type with absence support. The Yelp dataset is probably exported from a relational database, where foreign tables are inlined into the JSON object.

3.2.7. Conclusion

We have shown with four datasets, that a column can be constrained to one type with some additional support. The split produced by the schema is reasonable. In all four datasets, constraining 100% of the columns with (type/null)?, we still had an item coverage above 70%. This suggests that we can speed up the processing for the homogeneous, that makes more than two third of the data.

The additional column information speeds up the evaluation as the type information can be used to optimise the query and to get rid of the type checking overhead during processing.
4. Mapping

The goal of this chapter is to cover the theoretical aspect of the mapping from JSONiq to relational algebra. First, we redefine the JSONiq expression such that it can process items in parallel. We then give an overview how the mapping is done followed by the mappings.

4.1. Redefinition of Expressions

In the relational world a relational operator takes a table and transforms the table tuple-wise. The tuples are transformed in parallel. In JSONiq however, an expression transforms dynamic contexts one at a time. To have a similar behaviour in JSONiq, we redefine the JSONiq expression such that it can transform dynamic context in parallel. The redefinition simplifies later the mappings.

4.1.1. Sequence of Items as Dynamic Context

We can first observe that a sequence can be represented by a dynamic context by simply bind the sequence of items to a special variable $\$$.

For example, the following sequence

$$(1,2,\text{true})$$

can be represented by the following dynamic context.

$$< \$$ : (1,2,\text{true}) >$$

As described in Section 2.3.2, an expression is defined to be a function from dynamic contexts to sequences of items.

$$e : \mathcal{DC} \rightarrow \mathcal{S}$$

With our observation we can unifies the output and input of JSONiq expressions to dynamic contexts. We can rewrite the expression to a function that takes a dynamic context as input and outputs a dynamic context.

$$e : \mathcal{DC} \rightarrow \mathcal{DC}$$

1Whenever a dynamic context is actual using $\$$, we can use escape characters to substitute it.
4. Mapping

4.1.2. Collection-Oriented Expressions

In JSONiq an expression evaluates dynamic contexts one by another. In relational algebra however, the relational operator transforms all tuples in a relation in parallel. To close the difference, we would like to redefine the JSONiq expression such that it can take multiple dynamic contexts as input and output multiple dynamic contexts. Let us redefine an expression $e'$ to be

$$
\begin{pmatrix}
DC_{in}^1 \\
DC_{in}^2 \\
\vdots \\
DC_{in}^n
\end{pmatrix}
\overset{e'}{\rightarrow}
\begin{pmatrix}
DC_{out}^1 \\
DC_{out}^2 \\
\vdots \\
DC_{out}^n
\end{pmatrix}
$$

where our new defined expression transforms each incoming dynamic context $DC_{in}^i$ to $DC_{out}^i$ in parallel.

$$DC_{out}^i = e( DC_{in}^i ), \text{ for } 1 \leq i \leq n$$

Such a stream of dynamic contexts is called a dynamic context stream. We can simplify the JSONiq expression to a function which takes a dynamic context stream and returns a dynamic context stream.

$$e' : DCS \rightarrow DCS$$

In this thesis we use our redefined definition of a JSONiq expression $e'$ to match the processing nature of relational algebra.

4.2. Overview and Formal Definitions

4.2.1. Mapping Overview

To map expressions from JSONiq to the relational world we need two separate mappings:

1. JSONiq data model to relations mapping and
2. JSONiq expression to relational expressions mapping.

The goal of the first mapping is to find a function $\mu$ that takes a JSONiq dynamic context streams as input and outputs the relational representation of it. The goal of the second mapping is to find a function $\gamma$ that maps a JSONiq expression to relational operators, such that the relational operators transform the mapped dynamic context in an equivalent manner as the expression would transform the dynamic context. The two mappings are illustrated below in Figure 4.1.
On the left side, we have the JSONiq world. For processing, a JSONiq expression $e$ transforms an input dynamic context stream $DCS_{input}$ to $DCS_{output}$. In an almost homogeneous settingl we hope to get a speed up by processing the dynamic context stream with relational operators. For our approachl we need two mappings $\mu$ and $\gamma$. We first use the mapping $\mu$ to get a relational representation of the input dynamic context stream $DCS_{input}$. We call the representation $R_{input}$. Note that this representation can consist of mutiple tables. We then use the second mapping that maps the JSONiq expression $e$ to relational expressions $r$ where $r = \gamma(e)$. The relational expressions $r$ transforms the relational representation $R_{input}$ to $R_{output}$. $R_{output}$ possibly consists of multiple tables. We say the mapping $\gamma$ is correct if $R_{output}$ represents the dynamic context stream $DCS_{output}$.

### 4.2.2. Formal Definition

Before describing the two mappings, we start by formally defining the items and functions we are working with in the JSONiq and the relational world.

**Common Terminology**

**Strings** We denote the set of all strings with $S$.

**Partial function** A partial function $f$ between two sets $A$ and $B$ is a relation that does not associate any element of $A$ to more than an element of $B$:

$$\forall(x_A, x_B), (y_A, y_B) \in f : x_A = y_A \Rightarrow x_B = y_B$$

We denote the set of all partial function from $A$ to $B$ with $A \not\rightarrow B$. Rather than writing $(a, b) \in f$ we write $f(a) = b$. 

\[4.1\] 

**Figure 4.1.: Illustration of the two mappings $\mu$ and $\gamma$.**
4. Mapping

**JSONiq Items**

**Item** An item is a JSON building block which we denoted by $i \in I$. $I$ is the set of all items.

**Sequence of items** A sequence of items is denoted by $s \in S$, where $S$ describes the set of all sequences of zero or more JSON items.

**Dynamic context** We denote a dynamic context by $dc \in DC = S \not\rightarrow S$, where $DC$ is the set of all dynamic contexts. One dynamic context $dc$ is defined as a partial function from variable name (string) to sequences of items. $dc = \{key_1 \rightarrow value_1, \ldots, key_n \rightarrow value_n\}$, where $\forall i \text{ key}_i \in S, \forall i \text{ value}_i \in S^*$.

**Dynamic context stream** We denote a dynamic context stream by $dcs \in DCS$, where $DCS$ is the set of all dynamic context streams. One dynamic context stream $dcs$ is formally a sequence of zero or more dynamic contexts. $dcs = (dc_1, dc_2, \ldots, dc_n) \in DCS$, where $\forall i : dc_i \in DC$.

**Expression** An expression is denoted by $e \in E$, where $E$ is the set of all expressions.

**Clause** A clause is denoted by $c \in C$, where $C$ is the set of all clauses.

**Relation**

**Attribute** We denote a relational attribute by $A \in S$.

**Tuple** A tuple is denoted by $t : S \not\rightarrow V$, where $V$ is the set of all values.

**Relation** We denote a relation by $R = (t_1, t_2, \ldots, t_n)$, where $\forall i \text{ t}_i : S \not\rightarrow V$

4.3. JSONiq Data Mapping

The JSONiq Data Model as described in Subsection 2.3 contains two main concepts: sequences and dynamic contexts. Both are built on JSON items:

- objects
- arrays
- strings
- numbers
- booleans
- nulls

Note that for the sake of simplicity, we are excluding arrays for this thesis. The reasons behind this decision are:

1. Nested array data is already a field of studies for itself [7] [14].
2. We want to focus with this thesis on the heterogenous nature of JSON data.
We still support one level of nestedness with sequences of items in dynamic context. By excluding arrays from the JSON data mapping, we are left with atomic types and objects.

### 4.3.1. Relational Layout for DCS

To derive the relational layout for our dynamic context stream mapping, we start with a simple example of a dynamic context stream where sequences of items are restricted to only have at most one item. The following example is a dynamic context stream with two dynamic contexts, each with two variable bindings, $i$ and $j$.

\[
\begin{align*}
&(< i : 3, j : \text{true} >) \\
&(< i : 4, j : \text{false} >)
\end{align*}
\]

The goal is to map each dynamic context to a single tuple of a relation. We first observe that formally, a dynamic context is a partial function from strings to sequences of items.

\[
S \not\rightarrow S
\]

As we constraint sequences to single items, the partial function looks similar to the definition of a relational tuple, a partial function from strings to values. In our case the values are JSON items.

\[
S \not\rightarrow I
\]

The mapping from dynamic context to tuple is straightforward for atomic JSON values. It is the identity function. For the relational representation we additionally add a column named "cid" to identify the dynamic context. For our example we can map it to the following relation.

<table>
<thead>
<tr>
<th>cid</th>
<th>$i$</th>
<th>$j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>false</td>
</tr>
</tbody>
</table>

But how do we deal with objects? Considered that there are no arrays, we can always flatten an object to an object with only one level. To keep track of the origin of the values, we keep the path to the value in dot-notation as the key.\(^2\) For example, the next nested JSON object

```json
{
    "name" : "daniel",
    "study" : {
        "school" : "ETH",
        "discipline" : "Computer Science"
    }
}
```

\(^2\)If we already have fields using the dot, we can use escape characters to replace it.
can be flattened to

```
{
  "name": "daniel",
  "study.school": "ETH",
  "study.discipline" : "Computer Science"
}
```

To get rid of the objects, we first flatten all the objects in sequence. In a second step we
split each key-value pair in an object in a separate sequence for every key and use the
dot-notation of the path as key. For example, the dynamic context

```
$user : ({{"name" : "Tom","study" : "school" : "UZH"}}) >
```

can be rewritten to two separate sequences with the path as key

```
< $user.name : ("Tom"), $user.study.school : ("UZH") >
```

Using the same mapping as before, mapping dynamic contexts with sequences of size at
most one is straightforward. We use the following example dynamic context to highlight
the steps we discussed.

```
(user : ({{"name" : "Tom","study" : "school" : "UZH"}}) >
(user : ({{"name" : "Mat","age" : 25,"study" : "school" : "ETH"}}) >
```

We first flatten all the objects and then split the sequences based on the path names.

```
< $user.name : ("Tom"), $user.study.school : ("UZH") >
< $user.name : ("Mat"), $user.age : 25, $user.study.school : ("ETH") >
```

These two dynamic contexts can directly be mapped to a relation by using the key as
column name and the value as value. The resulting relation is shown below.

<table>
<thead>
<tr>
<th>cid</th>
<th>$user.name</th>
<th>$user.age</th>
<th>$user.study.school</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Tom&quot;</td>
<td>⊥</td>
<td>&quot;UZH&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;Mat&quot;</td>
<td>25</td>
<td>&quot;ETH&quot;</td>
</tr>
</tbody>
</table>

The question, how to support sequences of size greater than one, still remains. Se-
quences in JSONiq cannot be nested, therefore we only need to support one level of
"nesting", that is dynamic context with sequences of items. Our approach is to map
sequence of items to an external table. We use a reference value on both tables to know
which value belongs to which dynamic context. Consider the following dynamic contexts
as example. Each of them has two sequences of size two to three.

```
( (<$i : (1,2),   <$j : (true,true,false) >
( <$i : (3,4,5),   <$j : (true,true) >
```

40
is mapped to the relations below. We have three tables, one main table \( R^\text{main} \) to keep track of the dynamic contexts and two external tables \( R^\text{i} \) and \( R^\text{j} \) for the variables \( \$i \) and \( \$j \).

\[
\begin{array}{ccc}
\text{cid} & \$i_{\text{ref}} & \$j_{\text{ref}} \\
1 & \text{ref}_1 & \text{ref}_3 \\
2 & \text{ref}_2 & \text{ref}_4 \\
\end{array}
\]

\[
\begin{array}{ccc}
\$i_{\text{id}} & \$i \\
\text{ref}_1 & 1 \\
\text{ref}_2 & 3 \\
\text{ref}_4 & 4 \\
\end{array}
\]

\[
\begin{array}{ccc}
\$j_{\text{id}} & \$j \\
\text{ref}_3 & \text{true} \\
\text{ref}_4 & \text{true} \\
\end{array}
\]

The references from the main table to the external tables are arbitrary chosen. It can be replaced with any characters.

### 4.3.2. Example

To get a better understanding, we use the following dynamic context stream with two dynamic contexts

\[
\begin{align*}
&< \$n : (2), \quad \$users : \{\"name\" : "Tom", \"age\" : 25\}, \{\"name\" : "Peter"\} > \\
&< \$n : (25), \quad \$users : \{\"name\" : "Sarah", \"status\" : "single"\} >
\end{align*}
\]

First, we build the main table \( R^\text{main} \) with the following columns: \( \text{cid} \) as the context identifier and \( \$n_{\text{ref}}, \$users_{\text{ref}} \) for the variables.

\[
\begin{array}{ccc}
\text{cid} & \$n_{\text{ref}} & \$users_{\text{ref}} \\
1 & \text{ref}_1 & \text{ref}_3 \\
2 & \text{ref}_2 & \text{ref}_4 \\
\end{array}
\]

For each of the variables \( \$n, \$users \), we need an external table: \( R^\text{n} \) and \( R^\text{users} \).

The \( R^\text{n} \) table has exactly two columns, \( \$n_{\text{id}} \) and \( \$n \). \( R^\text{users} \) has more columns as it contains objects, \( \$users_{\text{id}}, \$users \) as every table and additionally \( \$users_{\text{name}}, \$users_{\text{age}}, \$users_{\text{status}} \) for the objects.

\[
\begin{array}{cc}
\$n_{\text{id}} & \$n \\
\text{ref}_1 & 2 \\
\text{ref}_2 & 25 \\
\end{array}
\]

\[
\begin{array}{cccc}
\$users_{\text{id}} & \$users & \$users_{\text{name}} & \$users_{\text{age}} & \$users_{\text{status}} \\
\text{ref}_3 & \bot & \text{Tom} & 25 & \bot \\
\text{ref}_3 & \bot & \text{Peter} & \bot & \bot \\
\text{ref}_4 & \bot & \text{Sarah} & \bot & \text{single} \\
\end{array}
\]

Each tuple in the external table represent one item in the sequence of items. With the references we assign the items to the right sequence in the corresponding dynamic context.
4.3.3. Inlined values

As in the example above, there are two places where we can put an atomic value, either directly in the main table or in an external table. To illustrate the two different approaches, we use the following dynamic context stream as an example.

\[
\begin{pmatrix}
< i : (2) > \\
< i : (true) > \\
< i : ("hello") >
\end{pmatrix}
\]

We can observe that in every dynamic context, the variable $i$ holds a sequence of only one atomic item. For the single atomic item, we have two ways to represent it with a relation. The first way is to treat the value of $i$ as a normal sequence. The representation needs two tables, one for the main table $R^{main}$ and one for the variable $i$ $R^{i}$.

\[
R^{main} = \begin{array}{|c|c|}
\hline
cid & i_ref \\
\hline
1 & ref_1 \\
2 & ref_2 \\
3 & ref_3 \\
\hline
\end{array} \quad R^{i} = \begin{array}{|c|c|}
\hline
i_id & i \\
\hline
ref_1 & 2 \\
ref_2 & true \\
ref_3 & "hello" \\
\hline
\end{array}
\]

The second approach is to directly inline the value into the main table, as the sequence only contains one atomic item. The representation needs only one table.

\[
R^{main} = \begin{array}{|c|c|}
\hline
cid & i \\
\hline
1 & 2 \\
2 & true \\
3 & "hello" \\
\hline
\end{array}
\]

For this thesis, we consider both representation as valid representation for the example dynamic context stream. We call the two tables, which represent the same dynamic context stream, equivalent. For the mapping however, we will mainly use the external table method.

4.3.4. Adapters

To map dynamic context streams to relation, we have two different approaches, the external table and the inlined method. In this section, we want to show that both representations can be transformed by using relational operators from one to another. We will call these functions adapters. Whenever there is a mismatch between the assumed and the actual representation, we can use the adapters to transform the table from one representation to another. Note that the adapters only work when the desired variable is a sequence of a single item.
4. Mapping

External-to-Inline Adapter

Consider the variable $i$ is in an external table $R^i_{\text{input}}$. Let $R^{main}_{\text{input}}$ be the main table. To inline the variable $i$ into the main table, we define the adapter as follow.

\[
\text{adapter}^{main}_{\text{ext} \rightarrow \text{in}} \left( R^{main}_{\text{input}}, R^i_{\text{input}} \right) := R^{output}_{\text{main}} = \pi_{i_{\text{ref}}, i_{\text{id}}} \left( R^{main}_{\text{input}} \bowtie_{i_{\text{ref}}=i_{\text{id}}} R^i_{\text{input}} \right)
\]

We illustrate the adapter with the following example tables.

<table>
<thead>
<tr>
<th>cid</th>
<th>$i_{\text{ref}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ref_1</td>
</tr>
<tr>
<td>2</td>
<td>ref_2</td>
</tr>
<tr>
<td>3</td>
<td>ref_3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i_{\text{id}}$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref_1</td>
<td>true</td>
</tr>
<tr>
<td>ref_2</td>
<td>true</td>
</tr>
<tr>
<td>ref_3</td>
<td>false</td>
</tr>
</tbody>
</table>

Using the adapter, our output table inlines the variable $i$ as desired.

<table>
<thead>
<tr>
<th>cid</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
</tr>
</tbody>
</table>

Inline-to-External Adapter

Consider the variable $i$ is already inlined in the main table $R^{main}_{\text{input}}$. To generate the new main table $R^{output}_{\text{main}}$ and external table $R^{output}_{i}$ for $i$, we need two adapter functions. $\text{adapter}^{main}_{\text{in} \rightarrow \text{ext}}$ returns the new main table and $\text{adapter}^{i}_{\text{in} \rightarrow \text{ext}}$ returns the new external table. We first define an intermediate table that holds the newly generated references.

\[
R_{\text{intermediate}} = \pi_{i_{\text{ref}}, i_{\text{id}}}(R^{main}_{\text{input}})
\]

We define the two adapters as follows.

\[
\text{adapter}^{main}_{\text{in} \rightarrow \text{ext}}(R^{main}_{\text{input}}) := R^{output}_{\text{main}} = \pi_{\{i\}}(R_{\text{intermediate}})
\]

\[
\text{adapter}^{i}_{\text{in} \rightarrow \text{ext}}(R^{main}_{\text{input}}) := R^{output}_{i} = \pi_{i_{\text{id}}}(R_{\text{intermediate}})
\]

4.4. JSONiq Query Mapping

In this section, we describe the second mapping that maps JSON expressions to relational operators. A catalogue of mappings is discussed in the next subsections.
4.4.1. Construction of Items - Atomic Literals

Expression description An atomic literal is an expression which returns a constant string, number, boolean or null for each dynamic context.

![Diagram of atomic literal expression]

Figure 4.2.: Illustration of an atomic literal expression.

Figure 4.2 illustrates an atomic literal expression where $\delta_{\text{output}}$ creates a $\text{DCS}_{\text{output}}$ with the atomic literal as $$, independently of the input. This is done for each dynamic context in $\text{DCS}_{\text{input}}$.

Expression example Let the atomic literal be "Tom". Consider the following input dynamic context stream with three dynamic contexts.

$$\text{DCS}_{\text{input}} = \begin{pmatrix}
< $i : (1) > \\
< $i : (2) > \\
< $i : (3) > 
\end{pmatrix}$$

The literal expression transforms the input the following way.

$$\begin{pmatrix}
< $i : (1) > \\
< $i : (2) > \\
< $i : (3) > 
\end{pmatrix} \xrightarrow{\delta_{\text{output}}} 
\begin{pmatrix}
< $$ : ("Tom") > \\
< $$ : ("Tom") > \\
< $$ : ("Tom") > 
\end{pmatrix}$$

Relational Example We start with the relational representation of $\text{DCS}_{\text{input}}$. The main input table is enough as the expression is ignoring the variables.

$$R^\text{main}_{\text{input}} = \begin{array}{cc}
cid & $i \\ 
1 & 1 \\
2 & 2 \\
3 & 3 
\end{array}$$

Our literal expression takes the input table and outputs a new table where the variable $$ is bound to the given literal. As literals are single items, we can assume that the variable $$ is inlined in the main table.
4. Mapping

\[ R_{\text{output}}^{\text{main}} = \begin{align*}
\text{cid} & \quad \text{$$} \\
1 & \quad "Tom" \\
2 & \quad "Tom" \\
3 & \quad "Tom"
\end{align*} \]

**Relational Operator** Let \( R_{\text{main}} \) be the main input which represents \( DC_{\text{Sinput}} \). We can assume that the main output table inlines the variable \( $$ \). The relational operator is defined as follows.

\[ \gamma_{\text{main}}(e)(R_{\text{main input}}^{\text{input}}, R_{\text{input var}_1}^{\text{input}}, \ldots, R_{\text{input var}_m}^{\text{input}}) := R_{\text{output}}^{\text{main}} = \pi_{\text{cid}, $$} \cdot \text{literal}(R_{\text{input}}^{\text{input}}) \]

### 4.4.2. Construction of Items - Object Construction

**Expression description** An object construction expression binds values to field variables.

Figure 4.3.: Illustration of an object construction expression.

Figure 4.3 illustrates an object construction expression. The delta functions \( \delta_1, \ldots, \delta_{2n} \) always return the first argument \( DC_{\text{Sinput}} \). \( \delta_{\text{output}} \) takes all the outputs from the subexpressions and creates a new object with \( n \) key-value pairs where the output of subexpression \( e_{2i-1} \) is the key and the output of subexpression \( e_{2i} \) is the value for \( 1 \leq i \leq n \).

**Expression example** Let the expression be \{\$key: $value, "fix": true\}. We use the following input dynamic context stream with two dynamic contexts:

\[
\begin{align*}
& <$key : ("key1"), $value : \{"name" : "Tom", "age" : 25\}> \\
& <$key : ("key2"), $value : \{"name" : "Marlen", "age" : 24\}> 
\end{align*}
\]

The four subexpressions that are black boxes for the expression return the following outputs.
$\delta_{\text{output}}$ takes the outputs and forms new objects with the output of $e_1$ being the keys for the values from $e_2$ and $e_3$ being the keys for the values from $e_4$. The resulting objects are the output $\mathcal{DCS}_{\text{output}}$.

$\delta_{\text{output}}(\mathcal{DCS}^1_{\text{output}}, \mathcal{DCS}^2_{\text{output}}, \mathcal{DCS}^3_{\text{output}}, \mathcal{DCS}^4_{\text{output}}) = \begin{cases} <\$\$ : \{"name" : "Tom", "age" : 25\} > \\
<\$\$ : \{"name" : "Marlen", "age" : 24\} > \end{cases}$

### Relational Example

The relational example starts with the mapped $\mathcal{DCS}_{\text{input}}$. In our example it is represented by three tables, the main table and two external tables for the variables $\$\$\$ key and $\$\$\$ value.

$$
\begin{array}{c|c|c|c|}
\text{R}_{\text{input}} & \text{cid} & \text{$\$\$ key_ref} & \text{$\$\$ value_ref} \\
\hline
\text{R}_{\text{main}} & 1 & \text{ref}\_1 & \text{ref}\_1 \\
\text{R}_{\text{main}} & 2 & \text{ref}\_2 & \text{ref}\_2 \\
\text{R}_{\$\$\$ key} & & \text{ref}\_1 & \text{\"key1\"} \\
\text{R}_{\$\$\$ key} & & \text{ref}\_2 & \text{\"key2\"} \\
\end{array}
\begin{array}{c|c|c|c|}
\text{R}_{\text{input}} & \text{\$\$ value_id} & \text{$\$\$ value} & \text{$\$\$ value.name} & \text{$\$\$ value.age} \\
\hline
\text{R}_{\$\$\$ value} & \text{ref}\_1 & \bot & \text{"Tom"} & 25 \\
\text{R}_{\$\$\$ value} & \text{ref}\_2 & \bot & \text{"Marlen"} & 24 \\
\end{array}
$$

The four subexpressions take the three tables as input and output the following tables. We can assume that the variable $\$\$ is inlined in the main tables, for two reasons. First, we do not allow arrays as values. Second, according to the specification we can treat the return values of the key subexpressions as atomic types.

$$
\begin{array}{c|c|}
\text{R}_{\text{output}} & \text{cid} & \$\$ \\
\hline
\text{R}_{\text{main}} & 1 & \text{"key1"} \\
\text{R}_{\text{main}} & 2 & \text{"key2"} \\
\end{array}
\begin{array}{c|c|c|}
\text{R}_{\text{output}} & \text{cid} & \$\$ \text{name} & \$\$ \text{.age} \\
\hline
\text{R}_{\text{main}} & 1 & \text{\"Tom\"} & 25 \\
\text{R}_{\text{main}} & 2 & \text{\"Marlen\"} & 24 \\
\end{array}
\begin{array}{c|c|}
\text{R}_{\text{output}} & \text{cid} & \$\$ \\
\hline
\text{R}_{\text{main}} & 1 & \text{true} \\
\text{R}_{\text{main}} & 2 & \text{true} \\
\end{array}
$$
δ\text{output} takes all the four tables from the subexpressions and outputs a new table that represents the mapped DC\text{S}\text{output}.

\[ R_{\text{output}} = \]

<table>
<thead>
<tr>
<th>cid</th>
<th>$$$</th>
<th>$$.key1.name$$</th>
<th>$$.key1.age$$</th>
<th>$$.key2.name$$</th>
<th>$$.key2.age$$</th>
<th>$$.fix$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\perp$</td>
<td>&quot;Tom&quot;</td>
<td>25</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>&quot;Marlen&quot;</td>
<td>24</td>
<td>true</td>
</tr>
</tbody>
</table>

**Relational Operator** Let \(R_{\text{main}}\) be the main table and \(R_{\text{input}}^{1}, \ldots, R_{\text{input}}^{m}\) be the external tables. All together represent \(DC\text{S}_{\text{input}}\). The relational operator for the main table is defined as follows. Note that we inline $$ into the main table.

\[
\gamma_{\text{main}}(e)(R_{\text{main}}, R_{\text{input}}^{1}, \ldots, R_{\text{input}}^{m}) := R_{\text{output}} = \\
\bigl(pivot_{cid,col,val}(\pi_{cid,col} \leftarrow prepend_{_\text{path}}(col,\$$),\text{val}(R_{\text{output},1} \bowtie_{cid} shred_{cid,col,val}(R_{\text{output},2}))\bigr) \bowtie_{cid} \\
\bigl(pivot_{cid,col,val}(\pi_{cid,col} \leftarrow prepend_{_\text{path}}(col,\$$),\text{val}(R_{\text{output},3} \bowtie_{cid} shred_{cid,col,val}(R_{\text{output},4}))\bigr) \bowtie_{cid} \\
\vdots \\
\bigl(pivot_{cid,col,val}(\pi_{cid,col} \leftarrow prepend_{_\text{path}}(col,\$$),\text{val}(R_{\text{output},2n-1} \bowtie_{cid} shred_{cid,col,val}(R_{\text{output},2n}))\bigr) \bowtie_{cid}
\]

where \(R_{\text{output},i}\) are the outputs from the subexpression \(e_i\)

\[
R_{\text{output},i} := \gamma_{\text{main}}(e_i)(R_{\text{main}}, R_{\text{input}}^{1}, \ldots, R_{\text{input}}^{m})
\]

The prepend_path function append the value from the second argument to the first argument after the $$. For example, prepend_path("$$\$$", "second") = $$\$$."second". In case there is just $$ we still append it right after the it prepend_path("$$", "second") = $$second".

Let us break the rather large relational algebra expression down. Let us have a look at the first line while using our example from before. The shred operator on the second subexpression main output table joined with the first subexpression main output table creates the following intermediate table.

<table>
<thead>
<tr>
<th>cid</th>
<th>col</th>
<th>val</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$$$$</td>
<td>&quot;Tom&quot;</td>
<td>&quot;key1&quot;</td>
</tr>
<tr>
<td>1</td>
<td>$$$$</td>
<td>25</td>
<td>&quot;key1&quot;</td>
</tr>
<tr>
<td>2</td>
<td>$$$$</td>
<td>&quot;Marlen&quot;</td>
<td>&quot;key2&quot;</td>
</tr>
<tr>
<td>2</td>
<td>$$$$</td>
<td>24</td>
<td>&quot;key2&quot;</td>
</tr>
</tbody>
</table>
The projection appends the variable $$ which represent the key in our key-value pair to the column name. The table after the projects is shown below.

<table>
<thead>
<tr>
<th>cid</th>
<th>col</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$$\text{key1.name}$$</td>
<td>&quot;Tom&quot;</td>
</tr>
<tr>
<td>1</td>
<td>$$\text{key1.age}$$</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>$$\text{key2.name}$$</td>
<td>&quot;Marlen&quot;</td>
</tr>
<tr>
<td>2</td>
<td>$$\text{key2.age}$$</td>
<td>24</td>
</tr>
</tbody>
</table>

With the `prepend_path` function, we exactly created the representation of the object we want. Now after a pivot operation the resulting table represents a portion of the final object.

<table>
<thead>
<tr>
<th>cid</th>
<th>$$\text{key1.name}$$</th>
<th>$$\text{key1.age}$$</th>
<th>$$\text{key2.name}$$</th>
<th>$$\text{key2.age}$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Tom&quot;</td>
<td>25</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>2</td>
<td>⊥</td>
<td>⊥</td>
<td>&quot;Marlen&quot;</td>
<td>24</td>
</tr>
</tbody>
</table>

For each pair of key-value subexpressions we get a portion of the final object. To create the final table which represent the final object, we need to join all the intermediate tables by cid.

### 4.4.3. Basic Operation - Comma Operator

**Expression description** A comma operator is a binary operator which takes two sequences of items from the subexpressions for each incoming dynamic context and concatenate them together. The output is a dynamic context with the concatenated sequences as $$.

![Figure 4.4. Illustration of a comma operator.](image)

Figure 4.4 illustrates a comma operator with its two subexpressions. The delta functions \( \delta_1 \) and \( \delta_2 \) always return the first argument \( DCS_{input} \). \( \delta_{output} \) creates the \( DCS_{output} \).
Expression example Let the expression be ("Bread", $i$). Consider the following input dynamic context stream with three dynamic contexts.

$$DCS_{input} = \begin{pmatrix} <i: \text{true}> \\ <i: \text{"Butter"}> \\ <i: (1,2,3)> \end{pmatrix}$$

With the following dynamic context stream as input, the output of $e_1$ and $e_2$ are as follows. Note that the subexpressions $e_1$ and $e_2$ are black boxes for the expression $e$.

$$
\begin{pmatrix}
<i: \text{true}>\\
<i: \text{"Butter"}>\\
<i: (1,2,3)>
\end{pmatrix} 
\overset{e_1}{\rightarrow} 
\begin{pmatrix}
<i: \text{true}>\\
<i: \text{"Butter"}>\\
<i: (1,2,3)>
\end{pmatrix} 
\overset{e_2}{\rightarrow} 
\begin{pmatrix}
<i: \text{true}>\\
<i: \text{"Butter"}>\\
<i: (1,2,3)>
\end{pmatrix}
$$

$\delta_{output}$ creates a new sequence of items by concatenating the two sequences from the output of $e_1$ and $e_2$ for each dynamic context.

$$
\begin{pmatrix}
<i: \text{"Bread"}>\\
<i: \text{"Butter"}>\\
<i: \text{"Bread"}>
\end{pmatrix} 
\overset{\delta_{output}}{\rightarrow} 
\begin{pmatrix}
<i: \text{"Bread"}, \text{true}>\\
<i: \text{"Bread"}, \text{"Butter"}>\\
<i: \text{"Bread"}, (1,2,3)>
\end{pmatrix}
$$

Relational example The relational example starts with the mapped $DCS_{input}$. In our example it is represented by two tables, the main table and an external table for the variable $i$.

$$R_{input}^{main} = \begin{pmatrix}
cid & i_ref \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3
\end{pmatrix} R_{input}^{i} = \begin{pmatrix}
$i_id$ & $i$ \\
\text{ref}_1 & \text{true} \\
\text{ref}_2 & \text{"Butter"} \\
\text{ref}_3 & 1 \\
\text{ref}_3 & 2 \\
\text{ref}_3 & 3
\end{pmatrix}$$

The two subexpressions take the two tables as input and output the following tables.

$$R_{output,1}^{main} = \begin{pmatrix}
cid & $i_id$ \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3
\end{pmatrix} R_{output,1}^{i} = \begin{pmatrix}
$i_id$ & $i$ \\
\text{ref}_1 & \text{true} \\
\text{ref}_2 & \text{"Butter"} \\
\text{ref}_3 & 1 \\
\text{ref}_3 & 2 \\
\text{ref}_3 & 3
\end{pmatrix}$$
4. Mapping

\[
R_{\text{main output},2} = \begin{array}{cc}
cid & $\_\text{ref} \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3
\end{array}
\]

\[
R^\$\_\text{output},2 = \begin{array}{cc}
$\_\text{id} & $ \\
\text{ref}_1 & \text{true} \\
\text{ref}_2 & "\text{Butter}" \\
\text{ref}_3 & 1 \\
\text{ref}_3 & 2 \\
\text{ref}_3 & 3
\end{array}
\]

\[\delta_{\text{output}}\] takes all the four tables from the subexpressions and outputs two new tables that represent the mapped DCS_{\text{output}}.

\[
R_{\text{main output}} = \begin{array}{cc}
cid & $\_\text{ref} \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3
\end{array}
\]

\[
R^\$\_\text{output} = \begin{array}{cc}
$\_\text{id} & $ \\
\text{ref}_1 & "\text{Bread}" \\
\text{ref}_1 & \text{true} \\
\text{ref}_2 & "\text{Bread}" \\
\text{ref}_2 & "\text{Butter}" \\
\text{ref}_3 & "\text{Bread}" \\
\text{ref}_3 & 1 \\
\text{ref}_3 & 2 \\
\text{ref}_3 & 3
\end{array}
\]

Relational Operator Let \(R_{\text{main input}}\) be the main input table and \(R^\$\text{inputvar}_1, \ldots, R^\$\text{inputvar}_n\) be the external tables. All together represent DCS_{\text{input}}. The relational operator for the main output table and external output table are defined as follows.

\[
\gamma_{\text{main}}(e)(R_{\text{main input}}, R^\$\text{inputvar}_1, \ldots, R^\$\text{inputvar}_n) = R_{\text{output}} = \pi_{\text{cid}, \_\text{ref} \rightarrow \_\text{id}}(R_{\text{main input}})
\]

\[
\gamma^{\$}$ (e)(R_{\text{main input}}, R^\$\text{inputvar}_1, \ldots, R^\$\text{inputvar}_n) =
\]

\[
\pi_{\_\text{id}, \_\text{ref}}(R_{\text{output}} \bowtie_{\text{cid}} (\pi_{\text{cid}, \_\text{ref} = \_\text{id}}(R_{\text{output}} \bowtie_{\text{cid}} (\pi_{\text{cid}, \_\text{ref} = \_\text{id}}(R_{\text{output}}) \cup (R_{\text{output}}) \))
\]

where \(R_{\text{output},i}\) and \(R^{\$}\text{output},i\) are the output tables from the subexpression \(e_i\).

\[
R_{\text{output},i} := \gamma_{\text{main}}(e_i)(R_{\text{input}}, R^\$\text{inputvar}_1, \ldots, R^\$\text{inputvar}_n)
\]

\[
R^{\$}\text{output},i := \gamma^{\$}$ (e_i)(R_{\text{input}}, R^\$\text{inputvar}_1, \ldots, R^\$\text{inputvar}_n)
\]
4.4.4. Basic Operation - Range Operator

**Expression description** A range expression is used to construct a sequence of consecutive integers for each incoming dynamic context. If either operand is an empty sequence, the result is an empty sequence. Otherwise it produces an increasing sequence of integers from the return value of subexpression $e_1$ to the return value of subexpression $e_2$.

![Figure 4.5: Illustration of a range operator.](image)

Figure 4.5: Illustration of a range operator.

Figure 4.5 illustrates a range operator with its two subexpressions. The two delta functions $\delta_1$ and $\delta_2$ always return the first argument $DCS_{input}$. $\delta_{output}$ creates a $DCS_{output}$ with the generated sequence of integers as $$ for each dynamic context.

**Expression example** Let the expression be $i$ to $j$. Consider the following input dynamic context stream $DCS_{input}$ with four dynamic contexts.

\[
\begin{align*}
&<i : (1), j : (3)> \\
&<i : (3), j : (1)> \\
&<i : (), j : (1)> \\
&<i : (5), j : (5)>
\end{align*}
\]

First, we use $DCS_{input}$ to get the output of the subexpressions $e_1$ and $e_2$. The subexpressions are black boxes for the expression $e$.

\[
\begin{align*}
&<i : (1), j : (3)> \\
&<i : (3), j : (1)> \\
&<i : (), j : (1)> \\
&<i : (5), j : (5)>
\rightarrow
\begin{align*}
&<i : (1)> \\
&<i : (3)> \\
&<i : ()> \\
&<i : (5)>
\end{align*}
\]

51
The output is computed by generating the sequences of items based on the outputs of \( e_1 \) and \( e_2 \).

\[
\begin{pmatrix}
< \$i : (1), \$j : (3) > \\
< \$i : (3), \$j : (1) > \\
< \$i : (), \$j : (1) > \\
< \$i : (5), \$j : (5) > \\
\end{pmatrix} \xrightarrow{e_2} \begin{pmatrix}
< \$i : (3) > \\
< \$i : (1) > \\
< \$i : (5) > \\
\end{pmatrix}
\]

**Relational Example** The relational example starts with the mapped \( \mathcal{DCS}_{\text{input}} \). In our example it is represented by three tables, the main table and two external tables for the two variables \$i and \$j.

\[
R_{\text{input}}^{\text{main}} = \begin{array}{ccc}
\text{cid} & \$i_{\text{ref}} & \$j_{\text{ref}} \\
1 & \text{ref}_1 & \text{ref}_1 \\
2 & \text{ref}_2 & \text{ref}_2 \\
3 & \text{ref}_3 & \text{ref}_3 \\
4 & \text{ref}_4 & \text{ref}_4 \\
\end{array}
\]

\[
R_{\text{input}}^{\$i} = \begin{array}{cc}
\$i_{\text{id}} & \$i \\
\text{ref}_1 & 1 \\
\text{ref}_2 & 3 \\
\text{ref}_4 & 5 \\
\end{array}, \quad R_{\text{input}}^{\$j} = \begin{array}{c}
\$j_{\text{id}} \\
\text{ref}_1 \\
\text{ref}_2 \\
\text{ref}_3 \\
\text{ref}_4 \\
\end{array}
\]

The two subexpressions take the three tables as input and output the following tables. We can assume that the variable \$$ is inlined in the main tables as the output must be a single integer or an empty sequence.

\[
R_{\text{output},1}^{\text{main}} = \begin{array}{c}
\text{cid} & \$$ \\
1 & 1 \\
2 & 3 \\
3 & \bot \\
3 & 5 \\
\end{array}, \quad R_{\text{output},2}^{\text{main}} = \begin{array}{c}
\text{cid} & \$$ \\
1 & 3 \\
2 & 1 \\
3 & 1 \\
3 & 5 \\
\end{array}
\]

\( \delta_{\text{output}} \) takes the two output tables and outputs two new tables that represent \( \mathcal{DCS}_{\text{output}} \).

\[
R_{\text{output}} = \begin{array}{ccc}
\text{cid} & \$$_{\text{ref}} \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3 \\
4 & \text{ref}_4 \\
\end{array}, \quad R_{\text{output}}^{\$$} = \begin{array}{cc}
\$$_{\text{id}} & \$$ \\
\text{ref}_1 & 1 \\
\text{ref}_1 & 2 \\
\text{ref}_1 & 3 \\
\text{ref}_4 & 5 \\
\end{array}
\]
Relational Operator Let $R_{\text{input}}$ be the main input table and $R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_n}$ be the external tables. All together represent $DCS_{\text{input}}$.

We first need to define the helper table $R_{\text{int}}$. $R_{\text{int}}$ is an infinite table with one column with the name $\$\$. This column contains an infinite number of integers, ordered in an increasing order.

The relational operator for the main output table and external output table are defined as follows.

$$
\gamma_{\text{main}}(e)(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_n}) = R_{\text{output}}^{\text{main}} = \pi_{\text{cid}, \$\_\text{ref} \rightarrow \$\_\text{id}}(R_{\text{main}}^{\text{main}})
$$

$$
\gamma_{\$\$}(e)(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_n}) = \\
\pi_{\$\_\text{id}, \$\$} \left( \\
\rho_{\$\_\text{ref} \rightarrow \$\_\text{id}} \left( R_{\text{output}}^{\text{main}} \Join_{\text{cid}} \left( \\
\pi_{\text{cid}, R_{\text{int}}, \$\$ \rightarrow \$\$} \left( R_{\text{int}} \Join R_{\text{main}}^{\text{main}}, \$\$ \leq R_{\text{int}}, \$\$ \right) \ 	ext{and} \ R_{\text{int}}, \$\$ \leq R_{\text{main}}^{\text{main}}, \$\$ \right) \right) \\
\right)
$$

where $R_{\text{output}, i}^{\text{main}}$ are the output tables from the subexpression $e_i$. We assume that the variable $\$\$ is already inlined in the main table.

$$
R_{\text{output}, i}^{\text{main}} := \gamma_{\text{main}}(e_i)(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_n})
$$

4.4.5. Basic Operation - Binary Arithmetic

Expression description A binary arithmetic takes the two outputs form the two subexpressions $e_1$ and $e_2$ and applies the operation defined by the operator on them for each incoming dynamic context. The operators are $\text{mod}$, $\text{idiv}$, $\text{div}$, $\times$, $+$ or $\text{-}$.
Figure 4.6 illustrates a binary arithmetic expression with its two subexpressions. The delta functions $\delta_1$ and $\delta_2$ always return the first argument $DCS_{input}$. $\delta_{output}$ applies the operator on the outputs of $e_1$ and $e_2$.

**Expression example** Let the expression be $i + 2$. We use the following input dynamic context stream with three dynamic contexts.

\[
\begin{pmatrix}
< i : (1) > \\
< i : (2) > \\
< i : () > 
\end{pmatrix}
\]

The two subexpressions that are black boxes for the expression return the following outputs.

\[
\begin{pmatrix}
< i : (1) > \\
< i : (2) > \\
< i : () > 
\end{pmatrix} \xrightarrow{e_1} \begin{pmatrix}
< $ : (1) > \\
< $ : (2) > \\
< $ : () > 
\end{pmatrix}
\]

\[
\begin{pmatrix}
< i : (1) > \\
< i : (2) > \\
< i : () > 
\end{pmatrix} \xrightarrow{e_2} \begin{pmatrix}
< $ : (2) > \\
< $ : (2) > \\
< $ : (2) > 
\end{pmatrix}
\]

$\delta_{output}$ takes the outputs from the subexpressions and applies the operator. In our case it adds the numbers together.

\[
\begin{pmatrix}
< $ : (1) > \\
< $ : (2) > \\
< $ : () > 
\end{pmatrix}, \begin{pmatrix}
< $ : (2) > \\
< $ : (2) > \\
< $ : (2) > 
\end{pmatrix} \xrightarrow{\delta_{output}} \begin{pmatrix}
< $ : (3) > \\
< $ : (4) > \\
< $ : () > 
\end{pmatrix}
\]
4. Mapping

Relational Example The relational example starts with the mapped DCS\textsubscript{input}. In our example it is represented by two tables, the main table and one external tables for the variable $i$.

\[
R_{\text{input}}^\text{main} = \begin{array}{c|c}
\text{cid} & \$i\_ref \\
\hline
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3 \\
\end{array} \quad R_{\text{input}}^\$i = \begin{array}{c|c}
\$i\_id & \$i \\
\hline
\text{ref}_1 & 1 \\
\text{ref}_2 & 2 \\
\end{array}
\]

The two subexpressions take the two tables as input and output the following tables. We can assume that the variable $$ is inlined in the main tables as the output must be a single number or an empty sequence.

\[
R_{\text{output},1}^\text{main} = \begin{array}{c|c}
\text{cid} & $$ \\
\hline
1 & 1 \\
2 & 2 \\
3 & \bot \\
\end{array} \quad R_{\text{output},2}^\text{main} = \begin{array}{c|c}
\text{cid} & $$ \\
\hline
1 & 2 \\
2 & 2 \\
3 & 2 \\
\end{array}
\]

$\delta_{\text{output}}$ takes both tables from the subexpressions and outputs a new table that represents the mapped DCS\textsubscript{output}. We can assume that the variable $$ is inlined in the main table as the result is again a single number or an empty sequence.

\[
R_{\text{output}}^\text{main} = \begin{array}{c|c}
\text{cid} & $$ \\
\hline
1 & 3 \\
2 & 4 \\
3 & \bot \\
\end{array}
\]

Relational Operator Let $R_{\text{input}}^\text{main}$ be the main input table and $R_{\text{input}}^\$\text{inputvar}_1, \ldots, R_{\text{input}}^\$\text{inputvar}_n$ be the external tables. All together represent DCS\textsubscript{input}. We assume that the variable $$ is inlined in the main output table. The main output table is then defined as follows.

\[
\gamma^\text{main}(e)(R_{\text{input}}^\text{main}, R_{\text{input}}^\$\text{inputvar}_1, \ldots, R_{\text{input}}^\$\text{inputvar}_n) := R_{\text{output}}^\text{main} = \pi_{\text{cid}, $$} R_{\text{output},1}^\text{main} \bowtie_{\text{cid}} R_{\text{output},2}^\text{main}
\]

where $op$ is the operator and

\[
R_{\text{output},1}^\text{main} = \gamma^\text{main}(e_1)(R_{\text{input}}^\text{main}, R_{\text{input}}^\$\text{inputvar}_1, \ldots, R_{\text{input}}^\$\text{inputvar}_n)
\]

\[
R_{\text{output},2}^\text{main} = \gamma^\text{main}(e_2)(R_{\text{input}}^\text{main}, R_{\text{input}}^\$\text{inputvar}_1, \ldots, R_{\text{input}}^\$\text{inputvar}_n)
\]

The variable $$ is inlined in the main table for the subexpressions.
4.6. Basic Operation - Unary Arithmetic

**Expression description** A unary arithmetic expression takes one output from the subexpression \( e_1 \) and applies the operation defined by the operator on it. There are two operators: + and -.

Figure 4.7.: Illustration of a unary arithmetic expression.

Figure 4.7 illustrate a unary arithmetic expression with a single subexpression. The delta function \( \delta_1 \) returns the first argument \( \mathcal{DCS}_{input} \). \( \delta_{output} \) applies the operator on the outputs of \( e_1 \) and returns it as $$.

**Expression example** Let the expression be \(-i\). We use the following input dynamic context stream with three dynamic contexts.

\[
\begin{pmatrix}
< i : (1) > \\
< i : (2) > \\
< i : () >
\end{pmatrix}
\]

First, we use \( \mathcal{DCS}_{input} \) to get the output of the subexpression \( e_1 \).

\[
\begin{pmatrix}
< i : (1) > \\
< i : (2) > \\
< i : () >
\end{pmatrix} \xrightarrow{e_1} \begin{pmatrix}
< i : (1) > \\
< i : (2) > \\
< i : () >
\end{pmatrix}
\]

The output is computed by applying the arithmetic operator on the output of \( e_1 \).

\[
\begin{pmatrix}
< i : (1) > \\
< i : (2) > \\
< i : () >
\end{pmatrix} \xrightarrow{\delta_{output}} \begin{pmatrix}
< i : (1) > \\
< i : (2) > \\
< i : () >
\end{pmatrix}
\]

**Relational Example** The relational example starts with the mapped \( \mathcal{DCS}_{input} \). In our example it is represented by two tables, the main table and one external tables for the variable $i$. 

56
4. Mapping

\[
R_{\text{input}}^{\text{main}} = \begin{array}{cc}
\text{cid} & \$_i\_\text{ref} \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3
\end{array}
\quad R_{\text{input}}^{\$_i} = \begin{array}{cc}
\$_i\_\text{id} & \$_i \\
\text{ref}_1 & 1 \\
\text{ref}_2 & -2
\end{array}
\]

The subexpression takes the two tables as input and outputs the following table. We can assume that the variable \$_i is inlined in the main table as it must be a single number or an empty sequence.

\[
R_{\text{output}}^{\text{main}} = \begin{array}{cc}
\text{cid} & \$_i \\
1 & 1 \\
2 & -2 \\
3 & \perp
\end{array}
\]

\[\delta_{\text{output}}\] takes the table and applies the operator on \$_i. We can assume that \$_i is inlined in the main table as the result is a single number or an empty sequence.

\[
R_{\text{output}}^{\text{main}} = \begin{array}{cc}
\text{cid} & \$_i \\
1 & -1 \\
2 & 2 \\
3 & \perp
\end{array}
\]

**Relational Operator** Let \(R_{\text{input}}^{\text{main}}\) be the main input table and \(R_{\text{input}}^{\$$, \ldots, R_{\text{input}}^{\$$} be the external tables. All together represent \(\text{DCS}_{\text{input}}\). The relational operator for the main output table is defined as follows. We assume that the variable \$_i is inlined in the main table.

\[
\gamma^{\text{main}}(e)(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\$$, \ldots, R_{\text{input}}^{\$$}) := R_{\text{output}}^{\text{main}} = \\
\pi_{\text{cid}, \$_i \leftarrow \$_i}(R_{\text{output}}^{\text{main},1})
\]

where \(op\) is the operator and \(R_{\text{output}}^{\text{main},1}\) is the output from the subexpression \(e_1\). We assume that the variable \$_i is inlined in the main table.

\[
R_{\text{output}}^{\text{main},1} := \gamma^{\text{main}}(e_1)(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\$$, \ldots, R_{\text{input}}^{\$$})
\]

4.4.7. Basic Operation - String Concatenation

**Expression description** A string concatenation expression takes the two string outputs from the two subexpressions \(e_1\) and \(e_2\) and concatenate them together for each incoming dynamic context.
Figure 4.8 illustrates a string concatenation expression. The delta functions $\delta_1$ and $\delta_2$ always return the first argument $DCS_{input}$ and $\delta_{output}$ concatenates the strings from the outputs of $e_1$ and $e_2$.

Expression example Let the expression be $\texttt{i || "World!"}$. Consider the following input dynamic context stream with three dynamic contexts.

$$
\begin{array}{l}
< \$i : ("Hello ") > \\
< \$i : ("Bye ") > \\
< \$i : () >
\end{array}
$$

The output of $e_1$ and $e_2$ are as follows. Note that the subexpressions $e_1$ and $e_2$ are black boxes for the expression $e$.

$$
\begin{array}{l}
< \$i : ("Hello ") > \\
< \$i : ("Bye ") > \\
< \$i : () > \rightarrow \\
< \$i : ("Hello ") > \\
< \$i : ("Bye ") > \\
< \$i : () > \rightarrow \\
< \$i : ("World!") > \\
< \$i : ("World!") > \\
< \$i : ("World!") >
\end{array}
$$

$\delta_{output}$ concatenates the strings from the outputs of $e_1$ and $e_2$.

$$
\begin{array}{l}
< \$\$ : ("Hello ") >, \\
< \$\$ : ("Bye ") >, \\
< \$\$ : () > \rightarrow \\
< \$\$ : ("Hello World!") >, \\
< \$\$ : ("World!") >, \\
< \$\$ : () > \rightarrow \\
< \$\$ : ("Bye World!") >, \\
< \$\$ : ("World!") >
\end{array}
$$
Relational Example  The relational example starts with the mapped $\mathcal{DCS}_{\text{input}}$. In our example it is represented by two tables, the main table and an external table for the variable $i$.

\[
R^\text{main}_{\text{input}} = \begin{array}{c|c}
\text{cid} & \$i\_ref \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3 \\
\end{array}
\quad R^i_{\text{input}} = \begin{array}{c|c}
\$i \_id & \$i \\
\text{ref}_1 & \text{"Hello "} \\
\text{ref}_2 & \text{"Bye "} \\
\end{array}
\]

The two subexpressions take the two tables as input and output the following tables. We can assume that the variable $$ is inlined in the main table.

\[
R^\text{main}_{\text{output},1} = \begin{array}{c|c}
\text{cid} & $$ \\
1 & \text{"Hello "} \\
2 & \text{"Bye "} \\
3 & \perp \\
\end{array}
\quad R^\text{main}_{\text{output},2} = \begin{array}{c|c}
\text{cid} & $$ \\
1 & \text{"World!"} \\
2 & \text{"World!"} \\
3 & \text{"World!"} \\
\end{array}
\]

$\delta_{\text{output}}$ takes both tables from the subexpressions and outputs a single table that represent the mapped $\mathcal{DCS}_{\text{output}}$. We can assume that the variable $$ is inlined in the main table.

\[
R^\text{main}_{\text{output}} = \begin{array}{c|c}
\text{cid} & $$ \\
1 & \text{"Hello World!"} \\
2 & \text{"Bye World!"} \\
3 & \text{"World!"} \\
\end{array}
\]

Note that empty sequences are treated like empty strings for concatenation.

Relational Operator  Let $R^\text{main}_{\text{input}}$ be the main input table and $R^\text{input}_{\text{input}_1}, \ldots, R^\text{input}_{\text{input}_n}$ be the external tables. All together represent $\mathcal{DCS}_{\text{input}}$. The relational operator for the main output table is defined as follows. We assume that the variable $$ is inlined in the main table.

\[
\gamma^\text{main}(e)(R^\text{input}_{\text{input}_1}, \ldots, R^\text{input}_{\text{input}_n}) := R^\text{main}_{\text{output}} = \\
\pi_{\text{cid}, $$ \leftarrow R^\text{input}_{\text{output},1}; R^\text{output}_{\text{output},2}}(R^\text{main}_{\text{output},1} \bowtie \text{cid} R^\text{output}_{\text{output},2})
\]

where $R^\text{main}_{\text{output},i}$ is the main output table from the subexpression $e_i$. We assume that the variable $$ is inlined in the main table.

\[
R^\text{main}_{\text{output},i} := \gamma^\text{main}(e_i)(R^\text{main}_{\text{input}}, R^\text{input}_{\text{input}_1}, \ldots, R^\text{input}_{\text{input}_n})
\]
4.4.8. Basic Operation - Comparison

Expression description A comparison expression takes the two outputs from the two subexpressions \( e_1 \) and \( e_2 \) and applies on them the defined comparison for each incoming dynamic context. The output is a DCS with the result of the comparisons. There are six types of comparison operator inspired by MongoDBs two-letter symbols: \( eq, ne, lt, le, gt, ge \).

![Comparison expression diagram](image)

Figure 4.9.: Illustration of a comparison expression.

Figure 4.9 illustrates a comparison operator with its two subexpressions. The delta functions \( \delta_1 \) and \( \delta_2 \) always return the first argument \( DCS_{input} \). \( \delta_{output} \) compares the outputs of \( e_1 \) and \( e_2 \) and outputs the result of the comparison.

Expression example Let the expression be \( \$i \ le \$j \). Consider the following input dynamic context stream with three dynamic contexts.

\[
DCS_{input} = \begin{pmatrix}
<i : (5), j : (1) > \\
<i : ("Hello"), j : ("World") > \\
<i : (()), j : (2) > \\
\end{pmatrix}
\]

The output of \( e_1 \) and \( e_2 \) are as follows. Note that the subexpressions \( e_1 \) and \( e_2 \) are black boxes for the expression \( e \).

\[
\begin{pmatrix}
<i : (5), j : (1) > \\
<i : ("Hello"), j : ("World") > \\
<i : (()), j : (2) >
\end{pmatrix} \xrightarrow{\delta_1} \begin{pmatrix}
<$$ : (5) > \\
<$$ : ("Hello") > \\
<$$ : () >
\end{pmatrix}
\]

\[
\begin{pmatrix}
<i : (5), j : (1) > \\
<i : ("Hello"), j : ("World") > \\
<i : (()), j : (2) >
\end{pmatrix} \xrightarrow{\delta_2} \begin{pmatrix}
<$$ : (1) > \\
<$$ : ("World") > \\
<$$ : (2) >
\end{pmatrix}
\]
4. Mapping

$\delta_{\text{output}}$ compares the outputs of $e_1$ and $e_2$ and returns a new dynamic context stream.

$$
\left(\begin{array}{c}
<$$ : (5) > \\
<$$ : ("Hello") >
\end{array}\right),
\left(\begin{array}{c}
<$$ : (1) > \\
<$$ : ("World") > \\
<$$ : (2) >
\end{array}\right) \xrightarrow{\delta_{\text{output}}} \left(\begin{array}{c}
<$$ : (false) > \\
<$$ : (true) > \\
<$$ : () >
\end{array}\right)
$$

Relational Example

The relational example starts with the mapped $\mathcal{DCS}_{\text{input}}$. In our example it is represented by three tables, the main table and two external tables for the variables $i$ and $j$.

$$
R_{\text{main input}}^{\text{input}} =
\begin{array}{ccc}
\text{cid} & \$i_{\text{ref}} & \$j_{\text{ref}} \\
1 & \text{ref}_1 & \text{ref}_1 \\
2 & \text{ref}_2 & \text{ref}_2 \\
3 & \text{ref}_3 & \text{ref}_3
\end{array}
$$

$$
R_{\text{i input}}^{i} =
\begin{array}{ccc}
\$i_{\text{id}} & \$i \\
\text{ref}_1 & 5 \\
\text{ref}_2 & \text{"Hello"}
\end{array}
$$

$$
R_{\text{j input}}^{j} =
\begin{array}{ccc}
\$j_{\text{id}} & \$j \\
\text{ref}_1 & 1 \\
\text{ref}_2 & \text{"World"} \\
\text{ref}_3 & 2
\end{array}
$$

The two subexpressions take the three tables as input and output the following tables. We can assume that the variable $$ is inlined in the main table.

$$
R_{\text{output},1}^{\text{main}} =
\begin{array}{ccc}
\text{cid} & $$ \\
1 & 5 \\
2 & \text{"Hello"} \\
3 & \bot
\end{array}
$$

$$
R_{\text{output},2}^{\text{main}} =
\begin{array}{ccc}
\text{cid} & $$ \\
1 & 1 \\
2 & \text{"World"} \\
3 & 2
\end{array}
$$

$\delta_{\text{output}}$ takes both tables from the subexpressions and outputs a single table that represents the mapped $\mathcal{DCS}_{\text{output}}$. We can assume that $$ is inlined in the main table as the output is a single boolean or an empty sequence.

$$
R_{\text{output}}^{\text{main}} =
\begin{array}{ccc}
\text{cid} & $$ \\
1 & \text{false} \\
2 & \text{true} \\
3 & \bot
\end{array}
$$

Relational Operator

Let $R_{\text{input}}^{\text{main}}$ be the main input table and $R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_n}$ be the external tables. All together represent $\mathcal{DCS}_{\text{input}}$. We assume that the
variable $$ is inlined in the main output table. The main output table is then defined as follows.

$$\gamma_{\text{main}}(e)\left(R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_n}\right) := R_{\text{output}}^{\text{main output}} = \pi_{\text{cid}, \text{op}R_1, \ldots, \text{op}R_2} \left(R_{\text{output},1}^{\text{main output}}, \ldots, R_{\text{output},2}^{\text{main output}}\right)$$

where op is the comparison operator and $$ is the main output table from the subexpression $$i. We assume that the variable $$ is inlined in the main table.

$$R_{\text{output},i}^{\text{main output}} := \gamma_{\text{main}}(e_i)\left(R_{\text{input}}^{\text{main output}_1}, \ldots, R_{\text{input}}^{\text{main output}_n}\right)$$

### 4.4.9. Basic Operation - Binary Logics

**Expression description** A binary logic expression takes the two outputs from the two subexpressions $$ and $$ and applies on them the defined binary logic for each incoming dynamic context. The output is a DCS with the result of the binary logic. There are two types of binary logic operator: or or and.

![Figure 4.10.: Illustration of a binary logic expression.](image)

Figure 4.10 illustrates a binary logic expression with its two subexpressions. The two delta functions $$ and $$ always return the first argument DCS_input. $$ applies the binary logic operation on the outputs of $$ and $$ and outputs the result of the logic operation.

**Expression example** Let the expression be $$i and $$j. Consider the following input dynamic context stream DCS_input with three dynamic contexts.

\[
\begin{align*}
< \text{i} : (true), \text{j} : (true) > \\
< \text{i} : (false), \text{j} : (true) > \\
< \text{i} : (), \text{j} : (true) >
\end{align*}
\]

62
First, we use $\mathcal{DCS}_{\text{input}}$ to get the output of the subexpressions $e_1$ and $e_2$.

\[
\begin{pmatrix}
< i : (true), j : (true) > \\
< i : (false), j : (true) > \\
< i : (), j : (true) >
\end{pmatrix}
\xrightarrow{e_1}
\begin{pmatrix}
< $\$: (true) > \\
< $\$: (false) > \\
< $\$: () >
\end{pmatrix}
\begin{pmatrix}
< i : (true), j : (true) > \\
< i : (false), j : (true) > \\
< i : (), j : (true) >
\end{pmatrix}
\xrightarrow{e_2}
\begin{pmatrix}
< $\$: (true) > \\
< $\$: (true) > \\
< $\$: (true) >
\end{pmatrix}
\]

$\delta_{\text{output}}$ takes the outputs from the subexpressions and applies the operator. In our case it computes the logical conjunction.

\[
\begin{pmatrix}
< $\$: (true) > \\
< $\$: (false) > \\
< $\$: () >
\end{pmatrix}
\xrightarrow{\delta_{\text{output}}}
\begin{pmatrix}
< $\$: (true) > \\
< $\$: (false) > \\
< $\$: () >
\end{pmatrix}
\]

**Relational Example** The relational example starts with the mapped $\mathcal{DCS}_{\text{input}}$. In our example it is represented by three tables, the main table and two external tables for the two variables $i$ and $j$.

<table>
<thead>
<tr>
<th>cid</th>
<th>$i_{\text{ref}}$</th>
<th>$j_{\text{ref}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ref_1</td>
<td>ref_1</td>
</tr>
<tr>
<td>2</td>
<td>ref_2</td>
<td>ref_2</td>
</tr>
<tr>
<td>3</td>
<td>ref_3</td>
<td>ref_3</td>
</tr>
</tbody>
</table>

$R_{\text{input}}^i$

<table>
<thead>
<tr>
<th>$i_{\text{id}}$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref_1</td>
<td>true</td>
</tr>
<tr>
<td>ref_2</td>
<td>false</td>
</tr>
</tbody>
</table>

$R_{\text{input}}^j$

<table>
<thead>
<tr>
<th>$j_{\text{id}}$</th>
<th>$j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref_1</td>
<td>true</td>
</tr>
<tr>
<td>ref_2</td>
<td>true</td>
</tr>
<tr>
<td>ref_3</td>
<td>true</td>
</tr>
</tbody>
</table>

The two subexpressions take the three tables as input and output the following tables. We can assume that the variable $\$$ is inlined in the main tables as the output must be a single boolean or an empty sequence.

$R_{\text{output,1}}^{\text{main}}$

<table>
<thead>
<tr>
<th>cid</th>
<th>$$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>3</td>
<td>⊥</td>
</tr>
</tbody>
</table>

$R_{\text{output,2}}^{\text{main}}$

<table>
<thead>
<tr>
<th>cid</th>
<th>$$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>true</td>
</tr>
</tbody>
</table>

$\delta_{\text{output}}$ takes the two output tables and returns a new table that represent $\mathcal{DCS}_{\text{output}}$. We can assume that the variable $\$$ is inlined in the main table it must be a boolean or an empty sequence.
4. Mapping

\[ R_{\text{output}} = \begin{array}{cc}
\text{cid} & \$\$
1 & \text{true}
2 & \text{false}
3 & \bot
\end{array} \]

**Relational Operator** Let \( R_{\text{input}}^{\text{main}} \) be the main input table and \( R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_n} \) be the external tables. All together represent \( \mathcal{DCS}_{\text{input}} \). We assume that the variable \$\$ is inlined in the main table. The main output table is then defined as follows.

\[
\gamma_{\text{main}}(\varepsilon)(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_n}) := R_{\text{output}} = \pi_{\text{cid}, \$\$ \leftarrow R_1, \$\$ \text{op} R_2, \$\$ (R_{\text{output}_{1}}, R_{\text{output}_{2}})
\]

where op is the binary logic operator and \( R_{\text{output}_{i}}^{\text{main}} \) are the output tables from the subexpression \( \varepsilon_i \). We assume that the variable \$\$ is already inlined in the main table.

\[
R_{\text{output}_{i}}^{\text{main}} := \gamma_{\text{main}}(\varepsilon_i)(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_n})
\]

**4.4.10. Basic Operation - Unary Logics**

**Expression description** A unary arithmetic expression takes the output from the subexpression \( \varepsilon_1 \) and applies the operation defined by the operator on it for each incoming dynamic context. There is just one unary operator: \textit{not}.

![Figure 4.11.: Illustration of a unary logic operator.](image)

Figure 4.11 illustrate a unary logic expression with a single subexpression. The delta function \( \delta_1 \) returns the first argument \( \mathcal{DCS}_{\text{input}} \). \( \delta_{\text{output}} \) applies the operator on the outputs of \( \varepsilon_1 \) and returns it as \$\$. 64
4. Mapping

**Expression example** Let the expression be not \(i\). We use the following input dynamic context stream with three dynamic contexts.

\[
\begin{pmatrix}
< i : (true) > \\
< i : (false) > \\
< i : () >
\end{pmatrix}
\]

First, we use \(\mathcal{DCS}_{input}\) to get the output of the subexpression \(e_1\).

\[
\begin{pmatrix}
< i : (true) > \\
< i : (false) > \\
< i : () >
\end{pmatrix} \xrightarrow{e_1} \begin{pmatrix}
< $$ : (true) > \\
< $$ : (false) > \\
< $$ : () >
\end{pmatrix}
\]

The output is computed by applying the logic operator on the output of \(e_1\).

\[
\begin{pmatrix}
< $$ : (true) > \\
< $$ : (false) > \\
< $$ : () >
\end{pmatrix} \xrightarrow{\delta_{output}} \begin{pmatrix}
< $$ : (false) > \\
< $$ : (true) > \\
< $$ : () >
\end{pmatrix}
\]

**Relational Example** The relational example starts with the mapped \(\mathcal{DCS}_{input}\). In our example it is represented by two tables, the main table and one external table for the variable \(i\).

\[
R_{i_{input}}^{main} = \begin{array}{c|c}
cid & $$_{i_{ref}} \\
1 & ref_1 \\
2 & ref_2 \\
3 & ref_3 \\
\end{array}
\]

\[
R_{i_{input}}^{SI} = \begin{array}{c|c}
$i_{id}$ & $i$ \\
ref_1 & true \\
ref_2 & false \\
\end{array}
\]

The subexpression takes the two tables as input and outputs the following table. We can assume that the variable $$ is inlined in the main table as it must be a single boolean or an empty sequence.

\[
R_{output,1}^{main} = \begin{array}{c|c}
cid & $$ \\
1 & true \\
2 & false \\
3 & \perp \\
\end{array}
\]

\(\delta_{output}\) takes the table and applies the not operator on $$. We can assume that $$ is inlined in the main table as the result is a single boolean or an empty sequence.

\[
R_{output}^{main} = \begin{array}{c|c}
cid & $$ \\
1 & false \\
2 & true \\
3 & \perp \\
\end{array}
\]
Relational Operator Let $R_{input}^{main}$ be the main input table and $R_{input}^{inputvar_1}, \ldots, R_{input}^{inputvar_{m}}$ be the external tables. All together represent $DCS_{input}$. The relational operator for the main output table is defined as follows. We assume that the variable $\$\$ is inlined in the main table.

\[
\gamma_{\text{main}}(e)(R_{input}^{main}, R_{input}^{inputvar_1}, \ldots, R_{input}^{inputvar_{m}}) := R_{output}^{main} = \Pi_{cid,\$\$\neg not R_{input}^{main.\$\$}(R_{output,1}^{main})
\]

where $R_{output,1}^{main}$ is the output from the subexpression $e_1$. We assume that the variable $\$\$ is inlined in the main table.

\[
R_{output,1}^{main} := \gamma_{\text{main}}(e_1)(R_{input}^{main}, R_{input}^{inputvar_1}, \ldots, R_{input}^{inputvar_{m}})
\]

4.4.11. Selector - Field Selector

Expression description A field selector navigates in an object in the same way as in JavaScript.

Figure 4.12.: Illustration of a field selector.

Figure 4.12 illustrates a field selector expression. The delta functions $\delta_1$ and $\delta_2$ always return the first argument $DCS_{input}$ and $\delta_{output}$ sets the output of $e_2$ as field name and treats it as string type. The value corresponding to the field name from $e_2$ is then extracted from the output of $e_1$ and returned as $\$\$.

Expression example Let the expression be $i$.field. Consider the following input
4. Mapping

dynamic context stream with three dynamic contexts.

\[
\begin{align*}
< \ & $i : 
    \begin{cases}
    \{ \text{Name}: "Dan", \text{Age}: 26 \}, & $field : \{\text{Name} \} > \\
    \{ \text{Name}: "Jose", \text{Name}: "Matt" \}, & $field : \{\text{Name} \} > \\
    \{ \text{Age}: "20" \}, & $field : \{\text{Age} \} > 
    \end{cases}
\end{align*}
\]

First, we use $DCS_{input}$ to get the output of the subexpressions $e_1$ and $e_2$.

\[
\begin{align*}
\begin{cases}
< \ & $i : 
    \begin{cases}
    \{ \text{Name}: "Dan", \text{Age}: 26 \}, & $field : \{\text{Name} \} > \\
    \{ \text{Name}: "Jose", \text{Name}: "Matt" \}, & $field : \{\text{Name} \} > \\
    \{ \text{Age}: "20" \}, & $field : \{\text{Age} \} > 
    \end{cases}
\end{cases} \\
\xrightarrow{e_1} \\
\begin{cases}
< \ & $$ : 
    \begin{cases}
    \{ \text{Name}: "Dan", \text{Age}: 26 \} > \\
    \{ \text{Name}: "Jose", \text{Name}: "Matt" \} > \\
    \{ \text{Age}: "20" \} > 
    \end{cases}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\begin{cases}
< \ & $i : 
    \begin{cases}
    \{ \text{Name}: "Dan", \text{Age}: 26 \}, & $field : \{\text{Name} \} > \\
    \{ \text{Name}: "Jose", \text{Name}: "Matt" \}, & $field : \{\text{Name} \} > \\
    \{ \text{Age}: "20" \}, & $field : \{\text{Age} \} > 
    \end{cases}
\end{cases} \\
\xrightarrow{e_2} \\
\begin{cases}
< \ & $$ : 
    \begin{cases}
    \{ \text{Name} \} > \\
    \{ \text{Name} \} > \\
    \{ \text{Age} \} > 
    \end{cases}
\end{cases}
\end{align*}
\]

$\delta_{output}$ then takes both outputs from $e_1$ and $e_2$ and returns the selected values as $$.

\[
\begin{align*}
\begin{cases}
< $$ : 
    \begin{cases}
    \{ \text{Name}: "Dan", \text{Age}: 26 \} > \\
    \{ \text{Name}: "Jose", \text{Name}: "Matt" \} > \\
    \{ \text{Age}: "20" \} > 
    \end{cases}
\end{cases} \\
\xrightarrow{\delta_{output}} \\
\begin{cases}
< $$ : 
    \begin{cases}
    \{ \text{Name} \} > \\
    \{ \text{Name}, \text{MAtt} \} > \\
    \{ \text{Age} \} > 
    \end{cases}
\end{cases}
\end{align*}
\]

Relational example The relational example starts with the mapped $DCS_{input}$. In our example it is represented by two tables, the main table and two external tables for the variables $i$ and $field$.

<table>
<thead>
<tr>
<th>cid</th>
<th>$i_{\text{ref}}$</th>
<th>$field_{\text{ref}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ref_1</td>
<td>ref_1</td>
</tr>
<tr>
<td>2</td>
<td>ref_2</td>
<td>ref_2</td>
</tr>
<tr>
<td>3</td>
<td>ref_3</td>
<td>ref_3</td>
</tr>
</tbody>
</table>
4. Mapping

\[
R^{\text{input}}_{i} = \begin{array}{cccc}
\text{i.id} & i & i.\text{name} & i.\text{age} \\
\text{ref}_1 & "\text{Dan}" & 20 \\
\text{ref}_2 & "\text{Jose}" & \downarrow \\
\text{ref}_3 & "\text{Matt}" & \downarrow \\
\end{array}
\]

\[
R^{\text{field}}_{\text{input}} = \begin{array}{cccc}
\text{field.id} & \text{field} \\
\text{ref}_1 & "\text{name}" \\
\text{ref}_2 & "\text{name}" \\
\text{ref}_3 & "\text{age}" \\
\end{array}
\]

The subexpressions take the two tables as input and output the following tables. We can assume that the variable $$ is inlined in the main table for the second subexpression as it must be a string.

\[
R^{\text{main}}_{\text{output},1} = \begin{array}{c|c|c|c|c|c}
\text{cid} & $$ \_\text{ref} \\
\hline
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3 \\
\end{array}
\]

\[
R^{\text{main}}_{\text{output},2} = \begin{array}{c|c|c|c|c|c|c}
\text{cid} & $$ \\
\hline
1 & "\text{name}" \\
2 & "\text{name}" \\
3 & "\text{age}" \\
\end{array}
\]

\[
\delta_{\text{output}} \text{ takes all tables and returns the desired tables.}
\]

\[
R^{\text{main}}_{\text{output}} = \begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\text{cid} & $$ \_\text{ref} & $$ \_\text{id} & $$ \_\text{col} & $$ \_\text{val} \\
\hline
1 & \text{ref}_1 & "\text{Dan}" & 20 \\
2 & \text{ref}_2 & "\text{Jose}" & \downarrow \\
3 & \text{ref}_3 & "\text{Matt}" & \downarrow \\
\end{array}
\]

Relational Operator Let \( R^{\text{main}}_{\text{output}} \) be the main input table and \( R^{\text{input}}_{\text{input}}, \ldots, R^{\text{input}}_{\text{input}} \) be the external tables. All together represent \( \mathcal{DCS}_{\text{input}} \). The relational operator for the main output table is defined as follows.

\[
\gamma^{\text{main}}(e)(R^{\text{main}}_{\text{input}}, R^{\text{input}}_{\text{input}}, \ldots, R^{\text{input}}_{\text{input}}) := R^{\text{main}}_{\text{output}} = R^{\text{main}}_{\text{output},1}
\]

\[
\gamma^{\text{input}}(e)(R^{\text{main}}_{\text{input}}, R^{\text{input}}_{\text{input}}, \ldots, R^{\text{input}}_{\text{input}}) := R^{\text{input}}_{\text{output}} = \text{pivot}^{\text{input}}_{\text{input}}(\pi_{\text{id}, \text{col}}(\pi_{\text{col}}(\sigma_{\text{col}}("\text{.}+\text{key}+"\_\text{col} = \text{remove}\_\text{path}(\text{col}, \text{val})\})

\rho_{\text{col}}("\text{.}+\text{key}+"\_\text{col}) \bowtie_{\text{id}} R^{\text{main}}_{\text{output},1}

\bowtie_{\text{ref} = \text{id} \_\text{col}} R^{\text{input}}_{\text{output},1})
\]
where $R_{main,1}^{output}$ and $R_{output,1}^{$$}$ are the outputs from the subexpression $e_1$. For $R_{output,2}^{main}$, we assume that the variable $$$ is inlined in the main table.

$$
R_{main,1}^{output} := \gamma_{main}(e_1)(R_{main,1}^{input}, R_{input}^{inputvar_1}, \ldots, R_{input}^{inputvar_m})
$$

$$
R_{output,1}^{$$} := \gamma_{$$}(e_1)(R_{main,1}^{input}, R_{input}^{inputvar_1}, \ldots, R_{input}^{inputvar_m})
$$

$$
R_{main,2}^{output} := \gamma_{main}(e_2)(R_{main,2}^{input}, R_{input}^{inputvar_1}, \ldots, R_{input}^{inputvar_m})
$$

The remove_path() function is the inverse function of append_path() from Subsubsection 4.4.2. It removes the key after $\$$$. For example, remove_path("$$$.key1.key2") = "$\$$$.key1".

For a better understanding we use the same example tables from above. The idea of the most inner join of the main tables $R_{main,1}^{output}$, $R_{output,2}^{main}$ and the external table $R_{output,2}^{$$}$ is to create a table where the key is inlined in the shredded external table. The following table illustrates the main information of the intermediate table.

<table>
<thead>
<tr>
<th>$$_id</th>
<th>col</th>
<th>val</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref_1</td>
<td>$$$.name</td>
<td>&quot;Dan&quot;</td>
<td>&quot;name&quot;</td>
</tr>
<tr>
<td>ref_1</td>
<td>$$$.age</td>
<td>20</td>
<td>&quot;name&quot;</td>
</tr>
<tr>
<td>ref_2</td>
<td>$$$.name</td>
<td>&quot;Jose&quot;</td>
<td>&quot;name&quot;</td>
</tr>
<tr>
<td>ref_2</td>
<td>$$$.name</td>
<td>&quot;Matt&quot;</td>
<td>&quot;name&quot;</td>
</tr>
<tr>
<td>ref_3</td>
<td>$$$.age</td>
<td>&quot;22&quot;</td>
<td>&quot;age&quot;</td>
</tr>
</tbody>
</table>

We then filter out the columns which are not interested in. This is done with a selection of the tuples where the col has the regex form $$_.key*$. Note that key is different for each tuple. The last step is to update the column name and to pivot the table back.

### 4.4.12. FLOWR Expressions

FLOWR expressions are "probably the most powerful JSONiq construct"[13]. On a theoretical level, it can be compared with the SELECT-FROM-WHERE statement of SQL, but FLOWR are "more general and more flexible"[13]. A FLOWR expression consists of multiple clauses. It must start with a for or let clause and end with a return clause. There are in total seven clauses:\n
- For clause
- Let clause
- Where clause
- Order by clause

\(^3\)The map expression is intentionally left out as it is only a shortcut for a for-return construct.
4. Mapping

- Group by clause
- Count clause
- Return clause

The communication between clauses happens with dynamic contexts. "A clause binds value to some variables according to its own semantics". Each time, a dynamic context "is passed on to the next clause. This goes all the way down, until the return clause" [13]. The return clause at the end returns a sequence of items for each dynamic context. These sequences of items are concatenated, in the order of the incoming dynamic context and the obtained sequence is returned by the FLOWR expression [13]. Figure 4.13 illustrates the passing of dynamic contexts in a general FLOWR expression.

A FLOWR expression starts with $\delta_1$ transforming the input $DC_{input}$ to a dynamic context stream $DCS_{input,1}$ with one dynamic context. All the subsequent delta functions $\delta_i$, $2 \leq i \leq n$, pass the last dynamic context stream $DCS_{output,(i-1)}$ to the next clause as input dynamic context stream $DCS_{input,i}$. The last delta function $\delta_{output}$ concatenates the sequences of items from the return clause.

Let us have a look how clauses work in detail. Clause $c_i$ is a function, that maps a dynamic context stream $DCS_{input}$ and outputs a new dynamic context stream $DCS_{output}$.

$$C : DCS \rightarrow DCS$$

The only exception is the return clause, which returns sequences of items $S_{output}$, one for each incoming dynamic context.

A clause can contain zero or more subexpressions. For each incoming dynamic context, the subexpressions are evaluated and based on the output of the subexpressions, new dynamic contexts are produced. Figure 4.14 illustrates a clause with its subexpressions.
Figure 4.14.: Illustration of a clause. The shortcuts in and out are used for input and output.

In a clause the delta functions $\delta_1, \ldots, \delta_n$ take each incoming dynamic context separately from the dynamic context stream and evaluate the subexpressions. Note that the subexpressions are evaluated multiple times, once for each dynamic context. $\delta_{\text{output}}$ takes $DC_{\text{input}}$ and all the output sequences and returns a new dynamic context $DC_{\text{output}}$. All these produced dynamic contexts are then bundled in $DC_{\text{output}}$. The only exception again is the return clause, where the $\delta_{\text{output}}$ returns directly the sequence of items from its subexpression. The semantic of $\delta_{\text{output}}$ depends on the type of the clause.

**Collection-based FLOWR Expressions and Clauses**

Analogously to the collection-based expressions, we want to redefine FLOWR expressions and clauses, such that it can process multiple input in parallel. Using the same argument as in Section 4.1 we can transform a FLOWR expression to a function that takes a dynamic context stream as input and outputs a dynamic context stream.

$$e : DCS \rightarrow DCS$$

As clauses are already functions communicating with dynamic context streams, we do not need to redefine them, except for the return clause. Using the same argument as above that we can always pack a sequence of items in a dynamic context, we can redefine the return clause to also communicate with a dynamic context stream. We still need to be careful with the return clause. We illustrate the problematic with an example. Consider the following example FLOWR expression.

for $i$ in (1 to $\max$)
return $i$

Let the input dynamic context stream consists of the following two dynamic contexts.

$$< \max : (2) >$$
$$< \max : (3) >$$
Figure 4.15 illustrates our example FLOWR expression.

With the dynamic context stream $DCS_{input}$, the for clause $c_1$ return the following dynamic context stream.

\[
\begin{align*}
&< i : (1), \ max : (2) > \\
&< i : (2), \ max : (2) > \\
&< i : (1), \ max : (3) > \\
&< i : (2), \ max : (3) > \\
&< i : (3), \ max : (3) > 
\end{align*}
\]

We can already observe that we have lost the information, which output dynamic context originate from which input dynamic context. This information is important for the delta function $\delta_{output}$ as our desired output dynamic context stream from the FLOWR expression is as follows.

\[
\begin{align*}
\left( < \$ : (1, 2) > \right) \\
\left( < \$ : (1, 2, 3) > \right)
\end{align*}
\]

Note that the first two dynamic contexts from the input dynamic context stream have been concatenated. The last three concatenated form the second dynamic context.

From the specification we know that the for clause is the only clause that can create for each incoming dynamic context multiple dynamic contexts as output. To remember the origin, we use a helper table that holds the information, which output dynamic context comes from which input dynamic context. For our example the table has the following form.
4. Mapping

<table>
<thead>
<tr>
<th>old_cid</th>
<th>new_cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

We assume that the dynamic contexts have a specific identifier called cid. With such a table the $\delta_{output}$ can map the dynamic context to their origin dynamic context. In case of multiple for clauses we can join the helper tables together to create a table from the input dynamic context identifiers to the current dynamic context identifiers.

$$\pi_{R_1.old\_cid,R_n.new\_cid} (R_1 \bowtie_{R_1.new\_cid=R_2.old\_cid} R_2 \cdots \bowtie_{R_{n-1}.new\_cid=R_n.old\_cid} R_n)$$

where $R_i$ is the helper table for the i-th for clause.

**FLOWR in-scope variables**

For the group by clause, we need to know if a particular variable comes from inside the FLOWR expression or outside. We call the variable $\$invar_i$ in the FLOWR scope if the variable has been created by one of the clauses in the current FLOWR expression without considering subexpressions. All the variables $\$outvar_j$ that came from the dynamic context stream $DCS_{input}$, we call them out of the FLOWR scope. To illustrate the concept and importance for the group by clause, consider the following nested FLOWR expression.

```flowr
let $seq := (1,1,2)
return (let $in := (1,1,2)
for $i in $seq
  group by $i
  return {
    "group key": $i,
    "out of scope": $seq,
    "in scope": $in
    }
  }
)
```

Note that the variable $\$in$ is in-scope for the inner FLOWR expression, while the variable $\$seq$ is out of scope. Let us have a look at the group by clause. The input dynamic context stream for the clause is as follows.

```
( < $i : (1), \$in : (1,1,2), \$seq : (1,1,2) > 
  < $i : (1), \$in : (1,1,2), \$seq : (1,1,2) > 
  < $i : (2), \$in : (1,1,2), \$seq : (1,1,2) > )
```
We can observe that the first two dynamic context have the same group key thus they are then grouped in the same group. The output dynamic context stream for the group by clause is as follows.

\[
\begin{align*}
    \langle i: (1), \text{in}: (1, 1, 2, 1, 2), \text{seq}: (1, 1, 2) > \\
    \langle i: (2), \text{in}: (1, 1, 2), \text{seq}: (1, 1, 2) >
\end{align*}
\]

Note that $\text{in}$ is in the output dynamic context concatenated while $\text{seq}$ is not. This is due to the FLOWR scope of the variables. The scope of the variable defines for non-group key variables if the value is concatenated or not.

FOR Clauses

Clause description A for clause evaluates the subexpression for each incoming dynamic context. Each item in the returned sequence $$ of the subexpression is then bound to the variable $\text{forname}$.

A for clause syntax has the form for $\text{forvar}$ in [subexpression $e_1$]. Figure 4.16 illustrates the for clause with the included subexpression where $\delta_1$ returns the first argument, $\text{DCS}_{\text{input}}$ and $\delta_{\text{output}}$ transforms each item in the returned sequence $$ from subexpression $e_1$ to an individual dynamic context.

Expression example Let the example for clause be for $i$ in $\text{seq}$. We use the following dynamic context stream as $\text{DCS}_{\text{input}}$.

\[
\begin{align*}
    \langle \text{seq}: (1, 2, 3) > \\
    \langle \text{seq}: (4) > \\
    \langle \text{seq}: (5, 6) >
\end{align*}
\]
First, we evaluate the subexpression \( e_1 \). The subexpression \( e_1 \) is for the clause a black box that returns a new dynamic context stream. In our case it returns a copy of \( \text{seq} \).

\[
\begin{pmatrix}
< \text{seq} : (1, 2, 3) > \\
< \text{seq} : (4) > \\
< \text{seq} : (5, 6) >
\end{pmatrix}
\xrightarrow{e_1}
\begin{pmatrix}
< \text{seq} : (1, 2, 3) > \\
< \text{seq} : (4) > \\
< \text{seq} : (5, 6) >
\end{pmatrix}
\]

Then \( \delta_{\text{output}} \) takes the \( DCS_{\text{input}} \) and the output of the subexpression \( e_1 \) and for each item in the sequence \( \$$ \) from the subexpression \( e_1 \), it returns a new dynamic context.

\[
\begin{pmatrix}
< \text{seq} : (1, 2, 3) > \\
< \text{seq} : (4) > \\
< \text{seq} : (5, 6) >
\end{pmatrix}
, \begin{pmatrix}
< \$$ : (1, 2, 3) > \\
< \$$ : (4) > \\
< \$$ : (5, 6) >
\end{pmatrix}
\xrightarrow{\delta_{\text{output}}}
\begin{pmatrix}
< \text{seq} : (1, 2, 3), \$$ : (1) > \\
< \text{seq} : (1, 2, 3), \$$ : (2) > \\
< \text{seq} : (1, 2, 3), \$$ : (3) > \\
< \text{seq} : (4), \$$ : (4) > \\
< \text{seq} : (5, 6), \$$ : (5) > \\
< \text{seq} : (5, 6), \$$ : (6) >
\end{pmatrix}
\]

**Relational example** The relational example starts with the mapped \( DCS_{\text{input}} \). In our example it is represented by two tables, the main table and one external table for the variable \( \text{seq} \).

\[
R_{\text{main}}^{\text{input}} = \begin{array}{c|c}
  cid & \text{seq_ref} \\
  \hline
  1 & \text{ref}_1 \\
  2 & \text{ref}_2 \\
  3 & \text{ref}_3 \\
\end{array}
\]

\[
R_{\text{seq}}^{\text{input}} = \begin{array}{c|c}
  \$$_id & \text{seq} \\
  \hline
  \text{ref}_1 & 1 \\
  \text{ref}_2 & 2 \\
  \text{ref}_3 & 3 \\
\end{array}
\]

The subexpression, that is a black box for the clause, returns a dynamic context stream. The mapped dynamic context stream is represented by two tables, the main table and external table for the variable \( \$$ \).

\[
R_{\text{main}}^{\text{output}} = \begin{array}{c|c}
  cid & \$$_ref \\
  \hline
  1 & \text{ref}_1 \\
  2 & \text{ref}_2 \\
  3 & \text{ref}_3 \\
\end{array}
\]

\[
R_{\$$}^{\text{output}, 1} = \begin{array}{c|c}
  \$$_id & \$$ \\
  \hline
  \text{ref}_1 & 1 \\
  \text{ref}_2 & 2 \\
  \text{ref}_3 & 3 \\
\end{array}
\]
The $\delta_{\text{output}}$ then takes all four tables and returns two tables, which represent the $\mathcal{DCS}_{\text{output}}$. From the specification of sequences of items, we know that it cannot be nested. As the for clause is unnesting the sequence of items, we can safely assume that we can inline the $\$\text{forvar}$ in the main table. The output for our example are two tables, the main table and the external table for the variable $\$\text{seq}$. The external table does not differ from the input table.

<table>
<thead>
<tr>
<th>cid</th>
<th>$i$</th>
<th>$\text{seq_ref}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>ref_1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>ref_1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>ref_1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>ref_2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>ref_3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>ref_3</td>
</tr>
</tbody>
</table>

**Relational Operator** Let $R_{\text{input}}^{\text{main}}$ be the main table and $R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m}$ be the external tables. All together represent $\mathcal{DCS}_{\text{input}}$. The relational operator for the main table is defined as follows. Note that we inline $\$\text{forvar}$ in the main table.

$$\gamma(c)_{\text{main}}(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m}) := R_{\text{output}}^{\text{output}} = \pi_{\text{cid} \leftarrow \text{id}()}(\rho_{\text{cid} \rightarrow \text{id}}(R_{\text{output}, 1}^{\text{output}} \bowtie_{\text{ref} = \$\text{seq\_ref}} R_{\text{output}, 1}^{\text{output}})))$$

where $R_{\text{output}, 1}^{\text{main}}$ and $R_{\text{output}, 1}^{\$\text{seq}}$ are the outputs from the subexpression $e_1$

$$R_{\text{output}, 1}^{\text{main}} := \gamma_{\text{main}}(e_1)(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m})$$

$$R_{\text{output}, 1}^{\$\text{seq}} := \gamma_{\$\text{seq}}(e_1)(R_{\text{input}}^{\text{main}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m})$$

For the external output tables, we copy them from the external input tables as the tables have not been changed. For each input variable $\$\text{inputvar}_i$, we have the following relational operator.

$$\gamma(c)_{\text{inputvar}_i}(R_{\text{input}}^{\text{inputvar}_i}, R_{\text{input}}^{\text{inputvar}_2}, \ldots, R_{\text{input}}^{\text{inputvar}_m}) := R_{\text{output}}^{\text{inputvar}_i} = R_{\text{input}}^{\text{inputvar}_i}$$

**Helper Table** As explained in Subsection 4.4.12 we need to keep the original dynamic contexts identifier for the return clause. The helper table is defined as follows.

$$R_{\text{helper}} = \pi_{\text{old\_cid} \leftarrow \text{new\_cid} \leftarrow \text{id}()}(\rho_{\text{cid} \rightarrow \text{old\_cid}}(R_{\text{output}, 1}^{\text{main}} \bowtie_{\$\text{ref} = \$\text{id}} R_{\text{output}, 1}^{\$\text{seq}})))$$
4. Mapping

LET Clauses

Clause description A let clause evaluates the subexpression for each incoming dynamic context. For each returned sequence $\$$ from the subexpression we bind it to the variable $\text{varname}$.

A let clause syntax has the form \texttt{let $\text{varname} := [\text{subexpression } e_1]$}. Figure 4.17 illustrates the let clause with the included subexpression $e_1$ where $\delta_1$ returns the first argument, $\text{DCS}_{\text{input}}$ and $\delta_{\text{output}}$ binds the output of subexpression $e_1$ to the variable $\text{varname}$.

Expression example Let the example let clause be \texttt{let $i := \text{old}$}. We use the following dynamic context stream as $\text{DCS}_{\text{input}}$.

\[
\begin{pmatrix}
< \text{old} : (1) > \\
< \text{old} : (2,3) > \\
< \text{old} : (4) >
\end{pmatrix}
\]

First, we evaluate the subexpression $e_1$. The subexpression $e_1$ is for the clause a black box, that returns a new dynamic context stream. In our case it returns a copy of $\text{old}$.

\[
\begin{pmatrix}
< \text{old} : (1) > \\
< \text{old} : (2,3) > \\
< \text{old} : (4) >
\end{pmatrix} \xrightarrow{e_1} 
\begin{pmatrix}
< \$$ : (1) > \\
< \$$ : (2,3) > \\
< \$$ : (4) >
\end{pmatrix}
\]

Then $\delta_{\text{output}}$ takes the $\text{DCS}_{\text{input}}$ and the output of the subexpression $e_1$ and renames the output of the subexpression from $\$$ to $\text{varname}$. 

77
4. Mapping

\[
\begin{pmatrix}
< \text{old} : (1) > \\
< \text{old} : (2, 3) > \\
< \text{old} : (4) >
\end{pmatrix}, \quad \begin{pmatrix}
< \text{$$} : (1) > \\
< \text{$$} : (2, 3) > \\
< \text{$$} : (4) >
\end{pmatrix} \xrightarrow{\delta_{\text{output}}} \begin{pmatrix}
< \text{old} : (1), \text{i} : (1) > \\
< \text{old} : (2, 3), \text{i} : (2, 3) > \\
< \text{old} : (4), \text{i} : (4) >
\end{pmatrix}
\]

Relational example The relational example starts with the mapped DCS\text{input}. In our example it is represented by two tables, the main table and one external table for the variable \text{old}.

\[
R_{\text{input}}^{\text{main}} = \begin{array}{cc}
\text{cid} & \text{old\_ref} \\
1 & \text{ref\_1} \\
2 & \text{ref\_2} \\
3 & \text{ref\_3}
\end{array} \quad R_{\text{input}}^{\text{old}} = \begin{array}{cc}
\text{old\_id} & \text{old} \\
\text{ref\_1} & 1 \\
\text{ref\_2} & 2 \\
\text{ref\_3} & 3 \\
\end{array}
\]

The subexpression, that is a black box for the clause, returns a dynamic context stream. The mapped dynamic context stream is represented by two tables, the main table and external table for the variable $$.

\[
R_{\text{output}, 1}^{\text{main}} = \begin{array}{cc}
\text{cid} & \text{$$$\_ref} \\
1 & \text{ref\_1} \\
2 & \text{ref\_2} \\
3 & \text{ref\_3}
\end{array} \quad R_{\text{output}, 1}^{\$$} = \begin{array}{cc}
\$$\_id & \$$ \\
\text{ref\_1} & 1 \\
\text{ref\_2} & 2 \\
\text{ref\_3} & 3 \\
\end{array}
\]

The \(\delta_{\text{output}}\) then takes all four tables and returns three tables, which represent the DCS\text{output}. The external table for the variable \text{old} does not differ from the input table.

\[
R_{\text{output}}^{\text{main}} = \begin{array}{ccc}
\text{cid} & \text{\_ref} & \text{old\_ref} \\
1 & \text{ref\_1} & \text{ref\_1} \\
2 & \text{ref\_2} & \text{ref\_2} \\
3 & \text{ref\_3} & \text{ref\_3}
\end{array} \quad R_{\text{output}}^{\text{old}} = \begin{array}{cc}
\text{old\_id} & \text{old} \\
\text{ref\_1} & 1 \\
\text{ref\_2} & 2 \\
\text{ref\_3} & 3 \\
\end{array} \quad R_{\text{output}}^{\text{i}} = \begin{array}{cc}
\text{i\_id} & \text{i} \\
\text{ref\_1} & 1 \\
\text{ref\_2} & 2 \\
\text{ref\_3} & 3 \\
\end{array}
\]

Relational Operator Let \(R_{\text{input}}^{\text{main}}\) be the main input table and \(R_{\text{input}}^{\text{input\_var_1}}, \ldots, R_{\text{input}}^{\text{input\_var_m}}\) be the external input tables. All together represent DCS\text{input}. The relational
operator for the main table and external tables are defined as follows.

\[
\gamma(c)^{\text{main}}(R_{\text{input}}^\text{main}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m}) := R_{\text{output}}^\text{main} \triangleleft (R_{\text{input}}^\text{main} \bowtie \Delta_{\text{id}}(\rho^{\text{varname}\rightarrow \text{inputvar}_i}(R_{\text{output}}^{\text{inputvar}_i}))
\]

\[
\gamma(c)^{\text{varname}}(R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m}) := R_{\text{output}}^{\text{varname}} = \rho^{\text{varname}\rightarrow \text{inputvar}_i}(R_{\text{output}}^{\text{inputvar}_i})
\]

where \(R_{\text{output,1}}^{\text{main}}\) and \(R_{\text{output,1}}^{\text{output}}\) are the outputs from the subexpression \(e_1\).

\[
R_{\text{output,1}}^{\text{main}} := \gamma^{\text{main}}(e_1)(R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m})
\]

\[
R_{\text{output,1}}^{\text{output}} := \gamma^{\text{output}}(e_1)(R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m})
\]

For the external output tables, we copy them from the external input tables as the tables have not been changed. For each input variable \(\text{inputvar}_i\), we have the following relational operator.

\[
\gamma(c)^{\text{inputvar}_i}(R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m}) := R_{\text{output}}^{\text{inputvar}_i} = R_{\text{input}}^{\text{inputvar}_i}
\]

**WHERE Clauses**

**Clause description** A where clause evaluates the subexpression for each incoming dynamic context as a boolean. If the returned boolean \$$ is false, we drop the dynamic context.

![Figure 4.18.: Illustration of a where clause.](image)

A where clause syntax has the form \textbf{where} [\text{subexpression} \(e_1\)]. Figure 4.18 illustrates the where clause with the included subexpression \(e_1\) where \(\delta_1\) returns the first argument, \(\text{DCS}_{\text{input}}\). \(\delta_{\text{output}}\) drops out the dynamic contexts based on the value \$$ returned by the subexpression \(e_1\).
Expression example Let the example where clause be where $i > 5$. We use the following dynamic context stream as $\mathcal{DCS}_{input}$.

\[
\begin{align*}
< i : (5) > \\
< i : (6) > \\
< i : (7) > 
\end{align*}
\]

First, we evaluate the subexpression $e_1$. The subexpression $e_1$ is for the clause a black box, that returns a new dynamic context stream. In our case it evaluates if $i$ is greater than 5.

\[
\begin{align*}
< i : (5) > & \mapsto < $\_$ : (false) > \\
< i : (6) > & \mapsto < $\_$ : (true) > \\
< i : (7) > & \mapsto < $\_$ : (true) > 
\end{align*}
\]

Then $\delta_{output}$ takes the $\mathcal{DCS}_{input}$ and the output of the subexpression $e_1$ and drops the dynamic context for which the subexpression returns a false for $\_$.

\[
\begin{align*}
< i : (5) > & \mapsto < i : (6) > \\
< i : (6) > & \mapsto < i : (7) > \\
< i : (7) > & \mapsto < i : (7) > 
\end{align*}
\]

Relational example The relational example starts with the mapped $\mathcal{DCS}_{input}$. In our example it is represented by two tables, the main table and one external table for the variable $i$.

\[
\begin{align*}
R_{input}^{main} & = \\
\begin{array}{cc}
\text{cid} & \text{$i$-ref} \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3 \\
\end{array} \\
R_{input}^{i} & = \\
\begin{array}{cc}
\text{$i$-id} & \text{$i$} \\
\text{ref}_1 & 5 \\
\text{ref}_2 & 6 \\
\text{ref}_3 & 7 \\
\end{array}
\end{align*}
\]

The subexpression, that is a black box for the clause, returns a dynamic context stream. The mapped dynamic context stream is represented by one table. From the specification, we know that the subexpression must return one boolean for each dynamic context. We can safely assume that the variable $\_$ is already inlined in the main table.

\[
\begin{align*}
R_{output,1}^{main} & = \\
\begin{array}{cc}
\text{cid} & \text{$\_$} \\
1 & \text{false} \\
2 & \text{true} \\
3 & \text{true} \\
\end{array}
\end{align*}
\]

The $\delta_{output}$ then takes all three tables and returns two tables, which represent the $\mathcal{DCS}_{output}$.
Relational Operator Let $R_{\text{main}}$ be the main input table and $R_{\text{inputvar}_1}, \ldots, R_{\text{inputvar}_m}$ be the external input tables. All together represent $\text{DCS}_{\text{input}}$. The relational operator for the main table and external tables are defined as follows.

$$
\gamma(c)^{\text{main}}(R_{\text{main}}^{\text{input}}, R_{\text{inputvar}_1}^{\text{input}}, \ldots, R_{\text{inputvar}_m}^{\text{input}}) := R_{\text{output}}^{\text{main}} = \\
\pi_{* / \{**\}} \left( \sigma_{\text{**}=\text{true}} \left( R_{\text{inputvar}_1}^{\text{input}} \bowtie_{\text{cid}} R_{\text{main}}^{\text{input}} \right) \right)
$$

$$
\gamma(c)^{\text{i}}(R_{\text{inputvar}_1}^{\text{input}}, \ldots, R_{\text{inputvar}_m}^{\text{input}}) := R_{\text{output}}^{\text{i}} = \\
\pi_{*_{\text{id}}, **_{\text{id}}} \left( \sigma_{\text{**}=\text{true}} \left( R_{\text{inputvar}_1}^{\text{input}} \bowtie_{\text{cid}} R_{\text{main}}^{\text{input}} \bowtie_{\text{cid}} R_{\text{outputvar}_n}^{\text{input}} \right) \right)
$$

where $i \in \{\text{inputvar}_1, \ldots, \text{inputvar}_n\}$. $R_{\text{output}, 1}^{\text{main}}$ and $R_{\text{output}, 1}^{\text{**}}$ are the outputs from the subexpression $e_1$.

$$
R_{\text{output}, 1}^{\text{main}} := \gamma^{\text{main}}(e_1)(R_{\text{inputvar}_1}^{\text{input}}, \ldots, R_{\text{inputvar}_m}^{\text{input}}) \\
R_{\text{output}, 1}^{\text{**}} := \gamma^{\text{**}}(e_1)(R_{\text{inputvar}_1}^{\text{input}}, \ldots, R_{\text{inputvar}_m}^{\text{input}})
$$

ORDER BY Clauses

Clause description An order by clause evaluates all the subexpressions for each incoming dynamic context and sorts the dynamic contexts lexicographically based on the output of the subexpressions.

An order by clause syntax has the form

$\delta_{\text{output}}$ ORDER BY Clause $c$

Figure 4.19.: Illustration of an order by clause.
order by [subexpression e₁] (ascending/descending), ..., [subexpression eₙ] (ascending/descending) (empty greatest/least)

Figure 4.19 illustrates the order by clause with the included subexpressions e₁, ..., eₙ. The delta functions δ₁, ..., δₙ always return the first argument, DCS\textsubscript{input}. The δ\textsubscript{output} sorts all dynamic contexts of DCS\textsubscript{input} based on the output of the subexpressions in a lexicographical order. The order for each subexpression can be specified in an ascending or descending order. Empty sequences can be attached at the beginning or at the very end.

**Expression example** Let the example order by clause be order by $i$ descending, $j$ ascending. We use the following dynamic context stream as DCS\textsubscript{input}.

\[
\begin{align*}
&\langle i : (1), j : (5) \rangle \\
&\langle i : (2), j : (1) \rangle \\
&\langle i : (2), j : (2) \rangle
\end{align*}
\]

First, we evaluate the subexpressions e₁ and e₂. The subexpressions are for the clause black boxes each returning a new dynamic context stream. In our case they return a copy of $i$ and $j$.

\[
\begin{align*}
&\langle i : (1), j : (5) \rangle \quad \xrightarrow{e₁} \quad \langle $i$ : (1), $j$ : (2) \rangle \\
&\langle i : (2), j : (1) \rangle \quad \xrightarrow{e₁} \quad \langle $i$ : (2), $j$ : (2) \rangle \\
&\langle i : (2), j : (2) \rangle \\
\end{align*}
\]

Then δ\textsubscript{output} takes the DCS\textsubscript{input} and the output of the subexpressions e₁ and e₂ and orders the dynamic contexts based on the output of the subexpressions lexicographically. Note that we order outputs of e₁ in a descending order and the outputs of e₂ in an ascending order.

\[
\begin{align*}
&\langle i : (1), j : (5) \rangle \quad , \quad \langle $i$ : (2), $j$ : (1) \rangle \\
&\langle i : (2), j : (2) \rangle \quad , \quad \langle $i$ : (2), $j$ : (2) \rangle \\
&\langle $i$ : (1), $j$ : (2) \rangle \\
\end{align*}
\]

Then δ\textsubscript{output} takes the DCS\textsubscript{input} and the output of the subexpressions e₁ and e₂ and orders the dynamic contexts based on the output of the subexpressions lexicographically. Note that we order outputs of e₁ in a descending order and the outputs of e₂ in an ascending order.

\[
\begin{align*}
&\langle i : (2), j : (1) \rangle \\
&\langle i : (2), j : (2) \rangle \\
&\langle $i$ : (1), $j$ : (2) \rangle \\
\end{align*}
\]

**Relational example** The relational example starts with the mapped DCS\textsubscript{input}. In our example it is represented by three tables, the main table and two external tables for the variables $i$ and $j$.
4. Mapping

\[
R_{\text{main input}} = \begin{array}{ccc}
\text{cid} & $i_{\text{ref}}$ & $j_{\text{ref}} \\
1 & \text{ref}_1 & \text{ref}_1 \\
2 & \text{ref}_2 & \text{ref}_2 \\
3 & \text{ref}_3 & \text{ref}_3 \\
\end{array}
\]

\[
R_{\text{i input}} = \begin{array}{cc}
\text{i id} & \text{i} \\
\text{ref}_1 & 1 \\
\text{ref}_2 & 2 \\
\text{ref}_3 & 2 \\
\end{array} \quad R_{\text{j input}} = \begin{array}{cc}
\text{j id} & \text{j} \\
\text{ref}_1 & 5 \\
\text{ref}_2 & 1 \\
\text{ref}_3 & 2 \\
\end{array}
\]

The subexpressions are black boxes for the clause. Each of the returned dynamic context stream is represented by one table. From the specification, we know that the subexpression must return an atomic or empty item. We can safely assume that the variable $$ is inlined in the main table. We denote the main output table of subexpression \(e_1\) as \(R_{\text{output,1}}^{\text{main}}\) and the main output table of subexpression \(e_2\) as \(R_{\text{output,2}}^{\text{main}}\).

\[
R_{\text{output,1}}^{\text{main}} = \begin{array}{cc}
\text{cid} & $$ \\
1 & 1 \\
2 & 2 \\
3 & 2 \\
\end{array} \quad R_{\text{output,2}}^{\text{main}} = \begin{array}{cc}
\text{cid} & $$ \\
1 & 5 \\
2 & 1 \\
3 & 2 \\
\end{array}
\]

The \(\delta_{\text{output}}\) then takes all five tables and returns three tables, one for the main table \(R_{\text{output}}^{\text{main}}\) and two external tables. These three tables represent the \(\text{DCS}_{\text{output}}\).
\(\text{DCS}_{\text{output}}\) is ordered in a lexicographical order based on the two tables \(R_{\text{output,1}}^{\text{main}}\) and \(R_{\text{output,2}}^{\text{main}}\).

\[
R_{\text{output}}^{\text{main}} = \begin{array}{ccc}
\text{cid} & $i_{\text{ref}}$ & $j_{\text{ref}} \\
1 & \text{ref}_2 & \text{ref}_2 \\
2 & \text{ref}_3 & \text{ref}_3 \\
3 & \text{ref}_1 & \text{ref}_1 \\
\end{array}
\]

\[
R_{\text{i output}}^{\text{output}} = \begin{array}{cc}
\text{i id} & \text{i} \\
\text{ref}_1 & 1 \\
\text{ref}_2 & 2 \\
\text{ref}_3 & 2 \\
\end{array} \quad R_{\text{j output}}^{\text{output}} = \begin{array}{cc}
\text{j id} & \text{j} \\
\text{ref}_1 & 5 \\
\text{ref}_2 & 1 \\
\text{ref}_3 & 2 \\
\end{array}
\]

**Relational Operator** Let \(R_{\text{input}}^{\text{main}}\) be the main input table and \(R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m}\) be the external input tables. All together represent \(\text{DCS}_{\text{input}}\). The relational
operator for the main table and external tables are defined as follows.

\[ \gamma(c)_{\text{main}} \left(R_{\text{input}}, R_{\text{input}1}, \ldots, R_{\text{input}m} \right) := R_{\text{output}} = \]

\[ \pi_{\text{cid} \leftarrow \text{id}, R_{\text{input}}, \text{input}1, \ldots, R_{\text{input}m} \left( \tau_{\text{ord}1 \pm \ldots \text{ord}n} \left( R_{\text{input}1} \bowtie R_{\text{output}1}, \right) \bowtie R_{\text{output}2}, \ldots \right) \]

where \( R_{\text{output},i} \) and \( R_{\text{output},i}^{\$} \) are the outputs from the subexpression \( e_i \)

\[ R_{\text{output},i} := \gamma_{\text{main}}(e_i) \left( R_{\text{input}}, R_{\text{input}1}, \ldots, R_{\text{input}m} \right) \]

\[ R_{\text{output},i}^{\$} := \gamma_{\text{input}}(e_i) \left( R_{\text{input}}, R_{\text{input}1}, \ldots, R_{\text{input}m} \right) \]

For the external output tables, we copy them from the external input tables as the tables have not been changed. For each input variable \( \text{input}var_j \), we have the following relational operator.

\[ \gamma(c)_{\text{input}var_j} \left(R_{\text{input}}, R_{\text{input}1}, \ldots, R_{\text{input}m} \right) := R_{\text{output}j} = R_{\text{input}j} \]

**COUNT Clauses**

**Clause description** A count clause adds a new variable to the incoming dynamic context, that binds the current ordinal position of the dynamic context in the dynamic context stream.

![Figure 4.20.: Illustration of a count clause.](image)

A count clause syntax has the form `count $varname`. Figure 4.20 illustrates the count clause. The delta functions \( \delta_{\text{output}} \) adds a new variable \$varname to the dynamic context that binds the current ordinal position.

**Expression example** Let the example count clause be `count $c`. We use the following dynamic context stream as \( \text{DCS}_{\text{input}} \).

\[
\left( < i : (true) > \right) \\
\left( < i : (false) > \right) \\
\left( < i : (false) > \right)
\]
4. Mapping

Then $\delta_{\text{output}}$ takes the $\mathcal{DCS}_{\text{input}}$ and adds a new variable that binds the original position of the dynamic context in the dynamic context stream.

\[
\begin{align*}
< & \$i : (true) > & \delta_{\text{output}} & \rightarrow & < & \$i : (true), & \$c : (1) > \\
< & \$i : (false) > & \delta_{\text{output}} & \rightarrow & < & \$i : (false), & \$c : (2) > \\
< & \$i : (false) > & \delta_{\text{output}} & \rightarrow & < & \$i : (false), & \$c : (2) > 
\end{align*}
\]

**Relational example** The relational example starts with the mapped $\mathcal{DCS}_{\text{input}}$. In our example it is represented by two tables, the main table and the external table for the variable $\$i$.

\[
\begin{align*}
R_{\text{input}}^\text{main} &= \begin{array}{|c|c|}
\hline
\text{cid} & \$i_{\text{ref}} \\
\hline
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3 \\
\hline
\end{array} & R_{\text{input}}^{\$i} &= \begin{array}{|c|c|}
\hline
\$i_{\text{id}} & \$i \\
\hline
\text{ref}_1 & \text{true} \\
\text{ref}_2 & \text{false} \\
\text{ref}_3 & \text{false} \\
\hline
\end{array}
\end{align*}
\]

The $\delta_{\text{output}}$ then takes the input tables and returns two tables, one for the main table $R_{\text{output}}^\text{main}$ and the external table for the variable $\$i$. From the specification we know that $\$c$ binds integers which are atomic items. We can assume that the variable is inline in the main table. Both tables together represent the $\mathcal{DCS}_{\text{output}}$.

\[
\begin{align*}
R_{\text{output}}^\text{main} &= \begin{array}{|c|c|c|}
\hline
\text{cid} & \$c & \$i_{\text{ref}} \\
\hline
1 & 1 & \text{ref}_1 \\
2 & 2 & \text{ref}_2 \\
3 & 3 & \text{ref}_3 \\
\hline
\end{array} & R_{\text{output}}^{\$i} &= \begin{array}{|c|c|}
\hline
\$i_{\text{id}} & \$i \\
\hline
\text{ref}_1 & \text{true} \\
\text{ref}_2 & \text{false} \\
\text{ref}_3 & \text{false} \\
\hline
\end{array}
\end{align*}
\]

**Relational Operator** Let $R_{\text{input}}^\text{main}$ be the main input table and $R_{\text{input}}^{\$inputvar_1}, \ldots, R_{\text{input}}^{\$inputvar_m}$ be the external input tables. All together represent $\mathcal{DCS}_{\text{input}}$. We assume that the variable $\$varname$ is inlined in the main table. The relational operator for the main table and external tables are defined as follows.

\[
\begin{align*}
\gamma(c)^{\text{main}}(R_{\text{input}}^\text{main}, R_{\text{input}}^{\$inputvar_1}, \ldots, R_{\text{input}}^{\$inputvar_m}) := R_{\text{output}}^\text{main} &= \pi_{\text{cid}, \$inputvar_1, \ldots, \$inputvar_m, \$varname-\text{id}}(R_{\text{input}}^\text{main})
\end{align*}
\]

For the external output tables, we copy them from the external input tables as the tables have not been changed. For each input variable $\$inputvar_j$, we have the following relational operator.

\[
\begin{align*}
\gamma(c)^{\$inputvar_j}(R_{\text{input}}^\text{main}, R_{\text{input}}^{\$inputvar_1}, \ldots, R_{\text{input}}^{\$inputvar_m}) := R_{\text{output}}^{\$inputvar_j} &= R_{\text{input}}^{\$inputvar_j}
\end{align*}
\]

**GROUP BY Clauses**

**Clause description** A group by clause evaluates all the subexpressions for each incoming dynamic context. The outputs of the subexpressions are the group keys. The
4. Mapping

dynamic contexts are then grouped according the group keys. For each group, a new dynamic context is returned.

A group by clause syntax has the following form.

```plaintext
group by $groupkey_1 := [subexpression e_1] , ..., $groupkey_n := [subexpression e_n]
```

Figure 4.21 illustrates the group by clause with the included subexpressions $e_1 ,..., e_n$. The delta functions $\delta_1, \ldots, \delta_n$ always return the first argument, $DCS_{input}$. The $\delta_{output}$ groups the incoming dynamic context based on the group keys. The group keys are the outputs from the subexpressions.

**Expression example** Let the example group by clause be $group by $name := $n, $num := $p$.

We use the following dynamic context stream as $DCS_{input}$. We assume that the three variables, $n$, $p$ and $seq$ are inside the FLOWR scope and $out$ is outside the scope. For the definition of FLOWR scope please refer to Subsection 4.4.12.

```
< $n : ("A"), $p : (1), $seq : (true, true), $out : "foo" >
< $n : ("A"), $p : (1), $seq : (false), $out : "foo" >
< $n : ("A"), $p : (2), $seq : (true, false), $out : "foo" >
< $n : ("B"), $p : (2), $seq : (true, false), $out : "bar" >
```

First, we evaluate the subexpressions $e_1$ and $e_2$. The subexpressions are for the clause a black boxes, each returning a new dynamic context stream. In our case they return a copy of $n$ and $p$.

```plaintext
< $n : ("A"), $p : (1), $seq : (true, true), $out : "foo" >
< $n : ("A"), $p : (1), $seq : (false), $out : "foo" >
< $n : ("A"), $p : (2), $seq : (true, false), $out : "foo" >
< $n : ("B"), $p : (2), $seq : (true, false), $out : "bar" >
```

86
The relational example starts with the mapped

\[
\begin{pmatrix}
< \$n : ("A"), & \$p : (1), & \$seq : (true, true), & \$out : "foo" > \\
< \$n : ("A"), & \$p : (1), & \$seq : (false), & \$out : "foo" > \\
< \$n : ("A"), & \$p : (2), & \$seq : (true, false), & \$out : "foo" > \\
< \$n : ("B"), & \$p : (2), & \$seq : (true, false), & \$out : "bar" > \\
\end{pmatrix}
\]

\(\xrightarrow{e_2}\)

\[
\begin{pmatrix}
< \$ : (1) > \\
< \$ : (1) > \\
< \$ : (2) > \\
< \$ : (2) > \\
\end{pmatrix}
\]

Then \(\delta_{\text{output}}\) takes the \(\text{DCS}_{\text{input}}\) and the output of the subexpressions \(e_1\) and \(e_2\) and groups the dynamic contexts based on the output of the subexpressions together.

\[
\begin{pmatrix}
< \$n : ("A"), & \$p : (1), & \$seq : (true, true), & \$out : "foo" > \\
< \$n : ("A"), & \$p : (1), & \$seq : (false), & \$out : "foo" > \\
< \$n : ("A"), & \$p : (2), & \$seq : (true, false), & \$out : "foo" > \\
< \$n : ("B"), & \$p : (2), & \$seq : (true, false), & \$out : "bar" > \\
\end{pmatrix}
\]

\(\xrightarrow{\delta_{\text{output}}}\)

\[
\begin{pmatrix}
< \$ : ("A") > \\
< \$ : ("A") > \\
< \$ : ("A") > \\
< \$ : ("B") > \\
\end{pmatrix}
\]

Note for the group keys \(\$name : ("A"), \$num : (1), \$n : ("A", "A"), \$p : (1, 1), \$seq : (true, true, false), \$out : "foo" > \)

\[
\begin{pmatrix}
< \$ : ("A") > \\
< \$ : ("A") > \\
< \$ : ("B") > \\
\end{pmatrix}
\]

Relational example The relational example starts with the mapped \(\text{DCS}_{\text{input}}\). In our example it is represented by three tables, the main table and four external tables for the variables \(\$n\), \(\$p\), \$seq and \$out.

\[
R_{\text{main}}^{\text{input}} =
\begin{array}{cccc}
\text{cid} & \$n_{\text{ref}} & \$p_{\text{ref}} & \$seq_{\text{ref}} & \$out_{\text{ref}} \\
1 & \text{ref}_1 & \text{ref}_1 & \text{ref}_1 & \text{ref}_1 \\
2 & \text{ref}_2 & \text{ref}_2 & \text{ref}_2 & \text{ref}_2 \\
3 & \text{ref}_3 & \text{ref}_3 & \text{ref}_3 & \text{ref}_3 \\
4 & \text{ref}_4 & \text{ref}_4 & \text{ref}_4 & \text{ref}_4 \\
\end{array}
\]

\[
R_{\text{input}}^{\$n} =
\begin{array}{ll}
\$n_{\text{id}} & \$n \\
\text{ref}_1 & "A" \\
\text{ref}_2 & "A" \\
\text{ref}_3 & "A" \\
\text{ref}_4 & "B" \\
\end{array}
\]

\[
R_{\text{input}}^{\$p} =
\begin{array}{ll}
\$p_{\text{id}} & \$p \\
\text{ref}_1 & 1 \\
\text{ref}_2 & 1 \\
\text{ref}_3 & 2 \\
\text{ref}_4 & 2 \\
\end{array}
\]

87
4. Mapping

\[ R^{seq}_{input} = \begin{array}{c|c}
seq_id & seq \\
\hline
ref_1 & true \\
ref_1 & true \\
ref_2 & false \\
ref_3 & true \\
ref_3 & false \\
ref_4 & true \\
ref_4 & false \\
\end{array} \]

\[ R^{out}_{input} = \begin{array}{c|c}
out_id & out \\
\hline
ref_1 & "foo" \\
ref_2 & "foo" \\
ref_3 & "foo" \\
ref_4 & "bar" \\
\end{array} \]

The subexpressions are black boxes for the clause. Each of the returned dynamic context stream is represented by one table. From the specification, we know that the subexpressions must return atomic items. We can safely assume that the variable \$ is inlined in the main table. We denote the main output table of subexpression \( e_1 \) as \( R^{main}_{output,1} \) and the main output table of subexpression \( e_2 \) as \( R^{main}_{output,2} \).

\[ R^{main}_{output,1} = \begin{array}{c|c}
cid & $$ \\
n & \hline
1 & "A" \\
2 & "A" \\
3 & "A" \\
4 & "B" \\
\end{array} \]

\[ R^{main}_{output,2} = \begin{array}{c|c}
cid & $$ \\
n & \hline
1 & 1 \\
2 & 1 \\
3 & 2 \\
4 & 2 \\
\end{array} \]

The \( \delta_{output} \) then takes all seven tables and returns five output tables, one for the main output table \( R^{main}_{output} \) and four external output tables. These five output tables represent the \( DCS_{output} \). The dynamic contexts in \( DCS_{output} \) are grouped based on the two tables \( R^{main}_{output,1} \) and \( R^{main}_{output,2} \).

\[ R^{main}_{output} = \begin{array}{c|c|c|c|c|c|c|c}
cid & $name & $num & $n_ref & $p_ref & $seq_ref & $out_ref \\
\hline
1 & "A" & 1 & ref_1 & ref_1 & ref_1 & ref_1 \\
1 & "A" & 2 & ref_2 & ref_2 & ref_2 & ref_2 \\
1 & "B" & 2 & ref_3 & ref_3 & ref_3 & ref_3 \\
\end{array} \]

\[ R^{n}_{output} = \begin{array}{c|c|c|c|c|c|c|c}
$n_id & $n \\
\hline
ref_1 & "A" \\
ref_1 & "A" \\
ref_2 & "A" \\
ref_3 & "B" \\
\end{array} \]

\[ R^{p}_{output} = \begin{array}{c|c|c|c|c|c|c|c}
$p_id & $p \\
\hline
ref_1 & 1 \\
ref_1 & 1 \\
ref_2 & 2 \\
ref_3 & 2 \\
\end{array} \]
Relational Operator  Let $R_{\text{input}}$ be the main input table and $R_{\text{input}1}, \ldots, R_{\text{input}m}$ be the external input tables. All together represent $DCS_{\text{input}}$. We denote $\$invar_i \in \{\$inputvar_1, \ldots, \$inputvar_m\}$ the variables that are in the FLOWR scope and $\$outvar_j \in \{\$inputvar_1, \ldots, \$inputvar_m\}$ the variables that are out of the FLOWR scope. $m_1$ denotes the number of variables in FLOWR scope and $m_2$ that are out of FLOWR scope. For the scope definition please refer to the Subsection 4.4.12. The relational operator for the main table is defined as follows.

$$
\gamma(c)_{\text{main}}(R_{\text{input}}^{R_{\text{input}1}}, \ldots, R_{\text{input}}^{R_{\text{input}m}}) := R_{\text{output}}^{R_{\text{output}}}
$$

$$
\pi_{\text{cid} \rightarrow \text{id}(\$groupkey_1, \ldots, \$groupkey_n), \$invar_1 \rightarrow \text{ref}_1(\ldots, \$invar_{m_1} \rightarrow \text{ref}_1(\ldots, \$outvar_{m_2} \rightarrow \text{ref}_1(\ldots}
$$

where $R_{\text{output},i}$ are the output from the subexpression $e_i$. We assume that the variable $\$$ is already inlined in the main table.

$$
R_{\text{output},i} := \gamma(\text{main}(e_i))(R_{\text{input}}, R_{\text{input}}^{R_{\text{input}1}}, \ldots, R_{\text{input}}^{R_{\text{input}m}})
$$

This relational algebra looks quite scary. Let us break it down. The middle part with all the joins by $cid$ creates a table where the group keys $\$groupkey_1, \ldots, \$groupkey_n$ is inlined into the main table. We then use the distinct operator to eliminate all the multiple instances as we want only one dynamic context for each group keys combination. We also add the variables which are out of scope, as we do not want to concatenate them. The next step is to choose the column we want to keep. These are all the group keys that are already inlined in the main table, all the variables in the FLOWR scope $\$invar_1, \ldots, \$invar_{m_1}$, and $m_2$ that get
new reference identifiers and all the variable reference out of the FLOWER scope $\texttt{outvar}_1\_\texttt{ref}, \ldots, $\texttt{outvar}_m\_\texttt{ref}$. The cid is then updated with new identifier.

For the external output tables of the variables $\texttt{outvar}_j$ that are out of scope, we copy them from the external input tables as the tables have not been changed.

$$
\gamma(c) \texttt{outvar}_j \left( R^{\texttt{main}}_{\texttt{input}}, R^{\texttt{inputvar}_1}_{\texttt{input}}, \ldots, R^{\texttt{inputvar}_m}_{\texttt{input}} \right) := R^{\texttt{outvar}_j}_{\texttt{output}} = R^{\texttt{outvar}_j}_{\texttt{input}}
$$

For each external output table of the variables $\texttt{invar}_i$, we have the following relational operator.

$$
\gamma(c) \texttt{invar}_i \left( R^{\texttt{main}}_{\texttt{input}}, R^{\texttt{inputvar}_1}_{\texttt{input}}, \ldots, R^{\texttt{inputvar}_m}_{\texttt{input}} \right) := R^{\texttt{output}}_{\texttt{inputvar}_j} = \pi_{\texttt{invar}_i\_\texttt{id}, \texttt{invar}_i\_\ast} ( \\
\rho_{\texttt{invar}_i\_\texttt{id} \leftarrow \texttt{invar}_i\_\texttt{ref} \left( \right.} \\
\left. P^{\texttt{main}}_{\texttt{input}} \bowtie_{ \texttt{groupkey}_1, \ldots, \texttt{groupkey}_n } ( \right.} \\
\left. R^{\texttt{input}}_{\texttt{invar}_i} \bowtie_{\texttt{cid}} \right.} \\
\left. P^{\texttt{main}}_{\texttt{input}} \bowtie_{\texttt{cid}} ( \rho_{\texttt{invar}_i\_\ast \leftarrow \texttt{groupkey}_1 \left( R^{\texttt{main}}_{\texttt{output}}, 1 \right)} \right.} \\
\left. \bowtie_{\texttt{cid}} ( \rho_{\texttt{invar}_i\_\ast \leftarrow \texttt{groupkey}_2 \left( R^{\texttt{main}}_{\texttt{output}}, 2 \right)} \right.} \\
\left. \vdots \right.} \\
\left. \bowtie_{\texttt{cid}} ( \rho_{\texttt{invar}_i\_\ast \leftarrow \texttt{groupkey}_n \left( R^{\texttt{main}}_{\texttt{output}}, n \right)} \right) \left. ) \right)

) \left. \right)
$$

Let us break down the rather big relational algebra expression. In the middle with all the join by cid, we build a table where we inline the group key into the $\texttt{invar}_i$ table. We then join the table with the main output table $R^{\texttt{main}}_{\texttt{output}}$ by the group keys $\texttt{groupkey}_1, \ldots, \texttt{groupkey}_n$. The resulting table inlines the newly generated references to the according $\texttt{invar}_i$. We then rename the column with the newly generated reference to $\texttt{invar}_i\_\texttt{id}$. The projection at the end selects the columns we are interested in, $\texttt{invar}_i\_\texttt{id}$ and all the values that represent $\texttt{invar}_i$.

**RETURN Clauses**

**Clause description** The return clause is the final clause of a FLWOR expression. It evaluates the subexpression $e_1$ for each incoming dynamic context and concatenates the results based on the helper table. The function of the helper table is to remember which dynamic context comes from which outer dynamic context. A for clause in a FLOWER expression can generate more dynamic contexts for each incoming dynamic. The return clause at the end must map all dynamic contexts
back to their original dynamic context. We assume that the helper table is already
the joined table (see 4.4.12).

![Diagram of return clause](image_url)

Figure 4.22.: Illustration of a return clause.

A return clause syntax has the form `return (subexpression e_1)`. Figure 4.22
illustrates the return clause. The delta function $\delta_{output}$ concatenates the results
based on the helper table.

**Expression example** Let the example return clause be `return $i$. We use the following
dynamic context stream as $DCS_{input}$.

\[
\begin{pmatrix}
< $i : (true) > \\
< $i : (false, false) > \\
< $i : (false) > \\
\end{pmatrix}
\]

We first evaluate the subexpression $e_1$ which takes the dynamic context stream as input and outputs the following dynamic context stream.

\[
\begin{pmatrix}
< $i : (true) > \\
< $i : (false, false) > \\
< $i : (false) > \\
\end{pmatrix} 
\xrightarrow{e_1} \begin{pmatrix}
< $$ : (true) > \\
< $$ : (false, false) > \\
< $$ : (false) > \\
\end{pmatrix}
\]

Then $\delta_{output}$ takes the output of the subexpression $e_1$ and based on the helper table, it concatenates the results together. Let the helper table be

<table>
<thead>
<tr>
<th>old_cid</th>
<th>new_cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

91
The return clause returns the following dynamic context stream as output.

\[
( < \$\$: (true, false, false) > \\
  < \$\$: (false) >
\]

**Relational example** The relational example starts with the mapped $\mathcal{DCS}_{input}$. In our example it is represented by two tables, the main table and the external table for the variable $\$_i$.

\[
\begin{array}{c|c}
\text{cid} & \$_i\_ref \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3 \\
\end{array}
\quad
\begin{array}{c|c}
\$_i\_id & \$_i \\
\text{ref}_1 & \text{true} \\
\text{ref}_2 & \text{false} \\
\text{ref}_3 & \text{false} \\
\end{array}
\]

The subexpression is a black box for the clause. We denote the main output table of subexpression $e_1$ with $R^\text{main}_{output,1}$ and the external table with $R^\$$_{output,1}$.

\[
\begin{array}{c|c}
\text{cid} & \$\$\_ref \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
3 & \text{ref}_3 \\
\end{array}
\quad
\begin{array}{c|c}
\$\$\_id & \$\$ \\
\text{ref}_1 & \text{true} \\
\text{ref}_2 & \text{false} \\
\text{ref}_3 & \text{false} \\
\end{array}
\]

Let the helper table be as follows.

\[
\begin{array}{c|c}
\text{old}\_cid & \text{new}\_cid \\
1 & 1 \\
1 & 2 \\
2 & 3 \\
\end{array}
\]

The $\delta_{output}$ then takes the helper table and the two output tables from the subexpression $e_1$ and returns two tables, the main table $R^\text{main}_{output}$ and the external table for the variable $\$\$$. Both tables together represent the $\mathcal{DCS}_{output}$.

\[
\begin{array}{c|c}
\text{cid} & \$\$\_ref \\
1 & \text{ref}_1 \\
2 & \text{ref}_2 \\
\end{array}
\quad
\begin{array}{c|c}
\$\$\_id & \$\$ \\
\text{ref}_1 & \text{true} \\
\text{ref}_1 & \text{false} \\
\text{ref}_2 & \text{false} \\
\end{array}
\]

**Relational Operator** Let $R^\text{main}_{input}$ be the main input table and $R^\text{inputvar}_1, \ldots, R^\text{inputvar}_m$ be the external input tables. All together represent $\mathcal{DCS}_{input}$. Let $R_{helper}$ be the
helper table. The relational operator for the main table and external tables are defined as follows.

\[
\gamma(c)_{\text{main}}(R_{\text{main}}^{\text{input}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m}):= R_{\text{output}}^{\text{output}} = \pi_{cid, \text{ref} \leftarrow \text{id}}(\rho_{old \_cid \rightarrow cid}(\text{distinct}_{old \_cid}(R_{\text{helper}))))
\]

\[
\gamma(c)_{\text{input}}(R_{\text{main}}^{\text{input}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m}):= \\
\pi_{\text{id}, \text{ref} \rightarrow \text{id}}(\rho_{\text{ref} \rightarrow \text{id}}( (R_{\text{output}}^{\text{output}} \bowtie_{old \_cid=\text{cid}} \pi_{old \_cid, \text{ref} \rightarrow \text{id}}(R_{\text{helper}} \bowtie_{new \_cid=\text{cid}} (R_{\text{output},2} \bowtie_{\text{id}=\text{id}} R_{\text{output},2}) ) ) ) )
\]

where \(R_{\text{output},1}^{\text{output}}\) and \(R_{\text{output},1}^{\text{input}}\) are the outputs from the subexpression \(e_1\)

\[
R_{\text{output},1}^{\text{output}} := \gamma_{\text{main}}(e_1)(R_{\text{main}}^{\text{main}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m})
\]

\[
R_{\text{output},1}^{\text{input}} := \gamma_{\text{input}}(e_1)(R_{\text{main}}^{\text{input}}, R_{\text{input}}^{\text{inputvar}_1}, \ldots, R_{\text{input}}^{\text{inputvar}_m})
\]
5. Optimisations

In this chapter we discuss the optimisation possibilities for the compiler. The goal is to give a first intuition about possible optimisations of our mapped relational algebra expression for an efficient processing. For the optimisation we have three level of optimisations in mind:

1. General relational algebra optimisation.
2. Optimisation with type inference from the query.
3. Optimisation with the data split.

5.1. Optimisation Example

To illustrate that the optimisations can transform a relational algebra down to an efficient query plan, we use a simple example. Consider the following JSONiq FLOWR expression.

```plaintext
for $i in (1,2)
return $i + 1
```

The recursive view of the expression is shown in Figure 5.1.

Using the relational mapping $\gamma$ described in Chapter 4, we can map the JSONiq FLOWR expression recursively to relational algebra. This result is shown in Figure 5.2.
The relational algebra might look large, but we show that it can be optimized with some rewriting rules to a much smaller query plan.

Figure 5.2.: Upper part of the relational algebra for our example JSONiq FLOWR expression.

First, we can observe a very common pattern, a transformation from the inlined repre-
presentation to the external representation followed by the reverse operation. This is due to the general assumption we make in the JSONiq expression mapping; we usually expect the input in external representation. We can get rid of such a pattern as they transform the tables from one representation to another and then back to the original representation. In Figure 5.2 we mark the part we can optimize in red. For the adapters in $e_4$, $e_5$ and $e_6$ we can first push the renaming and projection after the $Adapter_{ext\rightarrow in}^{main}$. Figure 5.3 illustrates the rewriting.

Figure 5.3.: The original query plan (left) and the rewritten query plan (right).

The renaming of $i\_ref$ to $$$\_ref$ and $i\_id$ to $$$\_id$ is not necessary as it is projected away in the $Adapter_{ext\rightarrow in}^{main}$. The projection $\pi_{cid, $$\_ref}$ is also redundant as the table has exactly the two columns. The only operator that we need to push down is the renaming of $i*$ to $$$*$. The adapters now resemble the pattern of splitting the table followed by a reverse operation. Such a pattern (marked in red) can be removed.

Using the optimisation of removing the redundant adapters, we get a new query plan depicted in Figure 5.4.
We show that the red operators in Figure 5.4 can be optimized.

For the upper red part, we first push the rename and projection down after the join. We need to be careful with the attribute $\$$ as it is present in both tables. The rewriting is illustrated in Figure 5.5.
We want to argue that the red join in the right query plan is redundant. We need for that three observations. First, in the left relations $R_l$ and $R_1$ the attributes $cid$ and $\_ref$ are unique for each tuple. Therefore, both joins do not create new tuples. Second, all the values in $R_r.cid$ are a subset of the values in $R_l.cid$. The same applies for the attribute $\_ref$ for the tables $R_2$ and $R_1$. That means the both join do not filter out tuples from the right tables $R_2$ and $R_r$. The third observation for the second join (the red one) is that all the attributes in the left relation $R_1$ is already in the right relation $R_2$. We can safely assume that the red join is redundant as it does not create nor remove tuples and all the attribute are already in the right relation $R_2$.

For the lower red part in Figure 5.4, we can push the rename and projection operator after the join by $cid$. We need to be careful with the variable $\_id$ as it exists in both relations. The rewriting is illustrated in Figure 5.6.

We know that the attribute $cid$ is unique for each tuple as it is newly generated by the $id(.)$ function. The self join can therefore be removed.

Using these two optimisations of removing the redundant joins, we get a new query plan depicted in Figure 5.7.
For the next optimisation step, we would like to have a closer look at the red and bold projection operator in Figure 5.7. It projects $cid$ and $i^*$. $cid$ is newly generated. That means $i^*$ is the only attribute of interest. We can push a projection up and remove all the (non-bold) operators in red as they are now redundant. The resulting query plan is depicted in Figure 5.8.
5. Optimisations

The two renaming operators in red can also be pushed up towards the beginning. For the three projections in red we can group them together to one projection. The resulting query plan is shown in Figure 5.9.¹

---

¹We can still argue to evaluate the calculation of $\$\$ + 1$ in $e_1$ and $e_2$. $e_1$ would directly output the literal $2$ and $e_2$ the literal $3$. 

---

Figure 5.8.: The query plan after the projection push-up.

Figure 5.9.: The query plan after merging projections and pushing up the rename operator.
adapter as glue code have been removed, and with common projection and rename push-ups, most of the redundant operators have been removed. Further research is needed to show that our mapping can be optimised in general. With this example we sketched the first few optimisation techniques.

5.2. Optimisation with Data Split

In Chapter 3 we introduced the data split depending on a chosen schema. The dataset $D$ is split into two parts, the homogeneous part $D_{homo}$ that is compliant to the given schema and the heterogeneous part $D_{hetero}$.

$$D = \tau(D_{homo} \cup D_{hetero})$$

where $\tau$ remembers and restores the ordering of the items. Using the mappings from Chapter 4 we can represent the dataset $D$ as a relation $R$ that is also split into two parts $R_{homo}$ and $R_{hetero}$.

$$R = \tau(R_{homo} \cup R_{hetero})$$

where $\tau$ remembers and restores the ordering of the items.

To illustrate the idea of the data split optimisation, we would like to start with an example JSONiq FLOWR expression that is similar to the one from last section. We add an external source `numbers.jsonl` which contains a sequence of numbers.

```json
for $i$ in json-file(numbers.jsonl)
  return $i + 1
```

Using the same optimisation and rewriting rules, we get the optimised query plan depicted in Figure 5.10. With $R_{numbers}$ we denote the table which is the mapped external source.

$$R_{numbers}$$

$\pi_{cid, id()}$; $+1$

```
Figure 5.10.: The optimised query plan for our example JSONiq FLOWR expression.
```

Our idea is to split the execution into two parts. We start at the top by replacing the relation $R_{numbers}$ with the split relations $\tau(R_{homo} \cup R_{hetero})$. For an efficient processing we want to push the union further down to separate the processing of the homogeneous and the heterogenous part. The schema and the additional type information helps the optimisation of the homogeneous part. For our example we know that the ordering
is commutative with selection and the selection is distributive under union. We can therefore derive the following rewrite rule to push the union further down.

\[ \pi_{col_1, col_2, \ldots, col_n}(\tau(R_{homo} \cup R_{hetero})) = \tau(\pi_{col_1, col_2, \ldots, col_n}(R_{homo}) \cup \pi_{col_1, col_2, \ldots, col_n}(R_{hetero})) \]

By applying the rewrite rule to our query plan can be rewritten as depicted in Figure 5.11.

![Figure 5.11: The optimised query plan with the data split.](image)

For the plus operation, we need to check on runtime for the data type of the variable \( \$$\$. As we know the type for the homogeneous part of the data from the schema, we can already check at compile time if the data fits the operation. The type check is only needed at runtime for the heterogenous part. This can be seen in Figure 5.11. At the red arrow, the tuples coming from \( R_{homo} \) do not need to be checked for the type at runtime. Only on the left side for the heterogenous part \( R_{hetero} \), a type check is needed.

We can see with our small example that the type checking overhead can be avoided during runtime by providing type information with a schema. Further research is needed for the optimisation part of the project.
6. Conclusion

Almost-homogeneous datasets are JSON datasets where a major part of the data is homogeneous. Relational database systems cannot handle the datasets as they still have a small heterogenous part. Systems built for semi-structured data on the other hand cannot optimise for the homogeneous part as they sacrifice efficiency for flexibility. But the flexibility is in practice not used to its full extent. The slowdown to process an almost-homogeneous semi-structured dataset is not justified.

In Chapter 3 we defined the homogeneity of a JSON dataset and proposed a theoretical mindset and framework to measure and visualize the homogeneity of a dataset in respect to the chosen schema. We applied the framework on four datasets and showed that in practice the datasets use the properties of JSON parsimoniously. The datasets are indeed mostly almost-homogeneous.

In order to build the envisioned system, we showed in Chapter 4 four main contributions and observations to successfully map the JSONiq world to relational algebra. We first redefined JSONiq expressions in a collection-oriented operator to match the parallel processing nature of relational algebra. We discussed sequences in relational algebra in Chapter 2 and introduced two new operators, *shred* and *pivot*, to enrich the relational algebra with NoSQL behaviours. The mapping from the JSONiq data model to relations has been defined, but we dropped the support for arrays as nestedness is a field of studies for itself. We also defined the mapping from JSONiq expressions to relational expressions, which can handle the recursion of the expression definition.

In Chapter 5 we sketched the possibility of optimisation for our mapped relational query plan. We showed an initial intuition of optimisation rules and hints. Using an example, we optimised the query plan with various rewriting rules. Besides normal relational optimisations the compiler in addition receives type information from the data split and from the query itself.

For future work, we propose further research in several directions.

**Array Support** In our mappings we excluded the arrays for simplicity. However, we support one level of nestedness with sequences of items. With arrays there could possibly be several levels of nesting. These nestings must be mapped to relations for example with multiple external tables.

**Type Support** JSONiq expressions can deal with types. A notion of *type* must be mapped into the representation of the items.

**Error Handling** In this thesis we always assumed that the queries are well-formed. JSONiq can throw errors at various stages and catch them during processing. These errors also need to be mapped to the relational world.
Optimisation  We only briefly introduced the optimisation of the mapped relational algebra. Further research is needed for the set of rewriting rules to achieve an optimal query plan.

Implementation  The development and implementation of the envisioned system in C++ using a JIT compiler is needed to show and measure the actual performance gain.
Bibliography


A. Appendix

A.1. Dataset Sources

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Source</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yelp</td>
<td><a href="https://www.yelp.com/dataset">https://www.yelp.com/dataset</a></td>
<td>business dataset, direct download.</td>
</tr>
</tbody>
</table>

A.2. Additional operators

In this section we list the operators that are either trivial to map or if they can be rewritten to other operators.

Variable Reference

The variable reference is straightforward. First, we choose the main table and the external table with the desired variable $\textit{varname}$. Then we do a renaming from the variable $\textit{varname}*$ to $$ followed by a projection. We project for the main table $\textit{cid}$ and $\textit{varname}_\textit{ref}$ and for the external table we project $\textit{varname}_\textit{id}$ and $\textit{varname}*$.  

Sequence Predicate

This can be rewritten to a FLOWR expression. If the predicate is an integer we rewrite it from the following expression

(expression e_1)[pred]
to the following FLOWR expression.

```
for $i$ in (expression e_1)
count $c$
where $c$ eq pred
return $i$
```

Otherwise we translate it to the following FLOWR expression.

```
for $i$ in (expression e_1)
where pred
return $i$
```

**Switch Expression**

A switch expression can be rewritten as an object construction followed by an object field selection. The following switch expression

```
switch (expression e)
case (expression c_1) return (expression r_1)
case (expression c_2) return (expression r_2)
...  
case (expression c_n) return (expression r_n)
```

can be rewritten to the following expression.

```
{  
  (expression c_1) : (expression r_1),
  (expression c_2) : (expression r_2),
  ...
  (expression c_n) : (expression r_n),
}.(expression e)
```

**Conditional Expression**

Conditional expressions are the binary case of switch expressions.

**A.3. Unnest expressions (Wrong approach)**

An expression can be nested, therefore for a clean mapping, we would like to "unnest" the inner expressions and apply the same rewriting in the same manner as before.

there are different kinds of inner expressions:

**Start expression** The start expression takes one dynamic context as input and produces a stream of items. It outputs the sequence of items alongside the same dynamic context from the input:

```
e_{start} : \mathcal{DC} \rightarrow \mathcal{DC} \times \mathcal{S}
```

A. Appendix

**Inner expression** An Inner expression takes two inputs: the dynamic context and possibly multiple stream of items. Please note that the dynamic context in an expression does not change, therefore it is enough to get only one dynamic context. The inner expression operates on the sequences of items with the dynamic context from the input and returns a new sequence of items and the same dynamic context.

\[ e_{\text{inner}} : \mathcal{DC} \times \mathcal{S} \times \cdots \times \mathcal{S} \rightarrow \mathcal{DC} \times \mathcal{S} \]

Some example of inner expressions are unary or binary expressions, with both they almost cover all the expressions in JSONiq.

**End expression** An end expression is conceptually the same as an inner expression, but it outputs only the new sequence of items and drops the dynamic context.

\[ e_{\text{end}} : \mathcal{DC} \times \mathcal{S} \times \cdots \times \mathcal{S} \rightarrow \mathcal{S} \]

To simplify all the different inner expressions, we use the notion of \( \mathcal{DC}? \) and \( \mathcal{S}^\ast \). The asterisk/plus sign matches the behaviour as in regex, thus \( \mathcal{DC}? \) means zero or one dynamic context and \( \mathcal{S}^\ast \) means zero or more sequences. Now all inner expressions can be expressed as:

\[ e_{\text{inner}} : \mathcal{DC} \times \mathcal{S}^\ast \rightarrow \mathcal{DC}? \times \mathcal{S} \]

We also use the batch definition, where the expression gets a vector of inputs.

\[ e_{\text{inner}} : \begin{pmatrix} \mathcal{DC} \\ \mathcal{DC} \\ \vdots \\ \mathcal{DC} \end{pmatrix} \times \left( \begin{pmatrix} \mathcal{S} \\ \vdots \end{pmatrix} \right)^\ast \rightarrow \begin{pmatrix} \mathcal{DC} \\ \vdots \end{pmatrix}? \times \begin{pmatrix} \mathcal{S} \\ \vdots \end{pmatrix} \]

As before we can argue that a sequence of items can be represented as a dynamic context by assigning it to a predefined variable name: $$$ . Now having everything as vector of dynamic contexts, we can simplify the inner expressions to:

\[ e_{\text{inner}} : \mathcal{DCS} \times \mathcal{DCS}^\ast \rightarrow \mathcal{DCS}? \times \mathcal{DCS} \]

The same can be seen on a syntax level. Let’s continue with the same example as before:

```plaintext
for $i$ in (1 to 5)
for $j$ in (1 to 5)
let $$ := $i \times 2 + $j$
let $z := $$$
return $z$
```

We now want to concentrate on the following nested (binary) expression \$i \times 2 + $j\. To unnest, we break the two arithmetic calculations into two clauses. The resulting query is:

109
for $i$ in (1 to 5)
for $j$ in (1 to 5)
let $\text{LHS} := i$
let $\text{RHS} := 2$
let $\text{LHS} := \text{LHS} \times \text{RHS}$
let $\text{RHS} := j$
let $$ := \text{LHS} + \text{RHS}$
let $z := $$
return $z$

After the rewriting the query only consists of clauses associated with only one non-nested expression. Now we only need to consider such clauses for the mapping.
Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

EFFICIENT PROCESSING OF ALMOST-HOMOGENEOUS SEMI-STRUCTURED DATA

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s): YU

First name(s): DANIEL

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zürich, 28.03.2018

Signature(s)