The Object-Relational Database Model-an Entry in noSQL database models

Dirk Van Gucht[1]

[1] Indiana University

March 9, 2019

Motivation

- In the relational database model, each relation has a schema of attributes with atomic domains such as booleans, numbers, text strings etc
- In the object-relational model, and in various noSQL data models, attributes in the schema the database may also have domains that consists of complex objects such as arrays, sets, bags, objects of composite types, relations, JSON objects, XML documents etc
- Therefore we need mechanisms to
  1. define relations/databases with attributes of complex-object types; and
  2. search and manipulate such relations/databases

Set and bag types as arrays

- In general, complex-object types can be recursively defined in terms of atomic types, composite types, array types, etc
- The main focus of this lecture will be on array types
- In particular, we will show how array types can be used to model bag and set types
- We will then show how operations on arrays allow us to model operations on bags and sets
- PostreSQL is an excellent system to consider the issues since it is an object-relational database system
- Many of the concepy discussed here can also be found in the noSQL MongoDB system as well as in the MapReduce framework and its derivatives

Arrays and the array constructor operation

- In SQL,

$$\text{`\{7, 4, 4, 3, 2\}'::int[]}$$

  denotes the array $[7, 4, 3, 3, 2]$ of type int[]
- Its first, third, and fifth component values are obtained as follows:

Array component	Value
('{7, 4, 4, 3, 2}'::int[])[1]	7
('{7, 4, 4, 3, 2}'::int[])[3]	4
('{7, 4, 4, 3, 2}'::int[])[5]	2

- In SQL,

$$\text{`\{``C'',``John'',``Anna'',``12''\}'::text[]}$$

  denotes the array ['C','John','Anna','12'] of type text[]
- Elements of an array must all be of the same type

Arrays and the array constructor operation (alternative syntax)

- In SQL, the following all denote the same array of integers

$$[7, 4, 3, 3, 2]$$

‘{7, 4, 4, 3, 2}’::int[]
ARRAY[7,4,4,3,2]::int[]
ARRAY[7,4,4,3,2]

- The third component value is obtained as follows

Array component	Value
(‘{7, 4, 4, 3, 2}’::int[])[3]	4
(ARRAY[7,4,4,3,2]::int[])[3]	4
(ARRAY[7,4,4,3,2])[3]	4

Modeling bags and sets with arrays

- The array

  $$\text{ARRAY[7,4,4,3,2]}$$

  represent (models) the bag

  $$\{2, 3, 4, 4, 7\}$$

  and the set

  $$\{2, 3, 4, 7\}$$

- Recall that an array orders its elements but a bag or a set does not
- The arrays ARRAY[7,4,4,3,2] and ARRAY[2,4,3,7,4] are different but they both represent the same bag and the same set
- The empty array '{}' or ARRAY[] models the empty set $\{\}$ (i.e., $\emptyset$)

ARRAY construction from a unary SQL query

- The ARRAY constructor operation can be applied to any SQL query that returns a unary relation
- It constructs an array of the elements of that relation

A

x
1
3
2
2

SELECT ARRAY(SELECT x FROM A)

$\rightarrow$

array

$\{1, 3, 2, 2\}$

A

x
1
3
2
2

SELECT ARRAY(SELECT DISTINCT x FROM A ORDER BY x)

$\rightarrow$

array

$\{1, 2, 3\}$

ARRAY construction from a SQL query with ROW construction

- The ARRAY constructor operation can be applied to any SQL query
- But, the tuples returned by the query must be packed to be of a composite (row) type by the ROW constructor operation

A

x	y
1	a
1	b
2	a

SELECT ARRAY(SELECT ROW(x,y) FROM A)

$\rightarrow$

SELECT ARRAY(SELECT (x,y) FROM A)

array

array
$\{(1, a), (1, b), (2, a)\}$

Example: A documents relation

- We may wish to maintain a database of documents and the words they contain
- We can define a relation with attributes of atomic domain to store such documents

```
CREATE TABLE documentWord (doc text, word text);
```

- A pair (d, w) in documentWord specifies that document d contains the word w

Example: A documents relation

- The documentWord relation may look as follow:

documentWord

doc	word
d1	A
d1	B
d1	C
d2	B
d2	C
d2	D
d3	A
d3	E
d4	B
d4	B
d4	A
d4	D
d5	E
d5	F
d6	A
d6	D
d6	G
d7	C
d7	B
d7	A
d8	B
d8	A

Example: The documents relation as a complex-objects relation

- We could consider a more natural representation of this data by having a relation documents of pairs (doc, words) where we pair each document with its set (bag) of words

documents

L

doc	words
d1	$\{A, B, C\}$
d2	$\{B, C, D\}$
d3	$\{A, E\}$
d4	$\{B, B, A, D\}$
d5	$\{E, F\}$
d6	$\{A, D, G\}$
d7	$\{C, B, A\}$
d8	$\{B, A\}$

- Such a relation is called a complex-objects relation
- We will see how the ARRAY type can be used to model such complex-objects relations
- We will then discuss how such relations can be manipulated and queried

The ARRAY type

- SQL has the array type
  1. for example, the array type text[] declares an array of text;
  2. int[] declares an array of int;
- SQL permits the use of these types in the definition of complex-object relations. For the documents relation, we can use the declaration

```
CREATE TABLE documents (doc text, words text[]);
```

- Such a table can be populated using insert statements such as

```
...
INSERT INTO documents VALUES ('d6', '{"A","D","G"}');
...
```

Querying the documents relation

- Next consider the query

  SELECT d.doc, d.words FROM documents d

- This query returns the contents of the documents relation
- The result would be

doc	words
d1	$\{A, B, C\}$
d2	$\{B, C, D\}$
d3	$\{A, E\}$
d4	$\{B, B, A, D\}$
d5	$\{E, F\}$
d6	$\{A, D, G\}$
d7	$\{C, B, A\}$
d8	$\{B, A\}$

Set and Bags as Unordered Array

- We will use arrays to represent sets (or bags).
- We must therefore restrict the predicates and operations we define on arrays to be independent of the order in which the elements appear in the arrays
- The following are such predicates and operations

$a \in A$	a is a member (element) of set A
$a \notin A$	a is not a member (element) of set A
$A \cap B \neq \emptyset$	set A and set B overlap
$A \subseteq B$	set A is a subset of set B
$A \supseteq B$	set A is a superset of set B
$A = \emptyset$	set A is empty
$\|A\|$	denotes the cardinality (size) of set A
$A \cup B$, $A \cap B$, $A - B$	union, intersection, difference of sets A and B

Checking for Set Membership ($a \in A$)

- In SQL this can be done using the $=$ SOME predicate
- “Find the documents that contain the word ‘D’ "

```
SELECT   d.doc, d.words
FROM     documents d
WHERE    ‘D’ = SOME(d.words)
```

doc	words
d2	$\{B, C, D\}$
d4	$\{B, B, A, D\}$
d6	$\{A, D, G\}$

Checking for Set Non-Membership ($a \notin A$)

- “Find the documents that do not contain the word ‘D’ "
- For this we can use the $<>$ ALL predicate.

```
SELECT   d.doc, d.words
FROM     documents d
WHERE    'D' <> ALL(d.words)
```

doc	words
d1	$\{A, B, C\}$
d3	$\{A, E\}$
d5	$\{E, F\}$
d7	$\{C, B, A\}$
d8	$\{B, A\}$

The isIn set-membership function

- For convenience, we define a polymorphic function isIn for the set-membership predicate:

```
CREATE FUNCTION isIn (x anyelement, A anyarray)
   RETURNS boolean AS
   $$
    SELECT x = SOME(A);
   $$ LANGUAGE SQL;
```

- We can now write the query “Find the documents that contain the word ‘A’ but not the word ‘D’ " as follows:

```
SELECT    d.doc, d.words
FROM      documents d
WHERE     isIn(‘A’,d.words) and not(isIn(‘D’,d.words))
```

Checking for Overlap of Sets ($A \cap B \neq \emptyset$)

- We may wish to check if sets overlap, i.e., if they have a non-empty intersection
- This can be done using the && predicate.
- “Find the documents whose sets of words overlap with the set of words $\{B, C\}$."

```
SELECT   d.doc, d.words
FROM     documents d
WHERE    d.words && ‘{“B",“C"}’
```

doc	words
d1	$\{A, B, C\}$
d2	$\{B, C, D\}$
d4	$\{B, B, A, D\}$
d7	$\{C, B, A\}$
d8	$\{B, A\}$

Checking for Disjoint (Non-overlapping) Sets ($A \cap B = \emptyset$)

- “Find the pairs of documents that do not have words in common."

```
SELECT   d1.doc AS doc1, d2.doc AS doc2,
         d1.words AS words1, d2.words AS words2
FROM     documents d1, documents d2
WHERE    NOT( d1.words && d2.words )
```

doc1	doc2	words1	words2
d1	d5	$\{A, B, C\}$	$\{E, F\}$
d2	d3	$\{B, C, D\}$	$\{A, E\}$
d2	d5	$\{B, C, D\}$	$\{E, F\}$
d3	d2	$\{A, E\}$	$\{B, C, D\}$
d4	d5	$\{B, B, A, D\}$	$\{E, F\}$
d5	d1	$\{E, F\}$	$\{A, B, C\}$
d5	d2	$\{E, F\}$	$\{B, C, D\}$
d5	d4	$\{E, F\}$	$\{B, B, A, D\}$
d5	d6	$\{E, F\}$	$\{A, D, G\}$
d5	d7	$\{E, F\}$	$\{C, B, A\}$
d5	d8	$\{E, F\}$	$\{B, A\}$
d6	d5	$\{A, D, G\}$	$\{E, F\}$
d7	d5	$\{C, B, A\}$	$\{E, F\}$
d8	d5	$\{B, A\}$	$\{E, F\}$

Checking for Set Containment (subset) ($A \subseteq B$)

- We may wish to check if a set is a subset of another set
- This can be done using the $<@$ set-containment predicate
- "Find the documents that contain the words 'A' and 'B' "

```
SELECT   d.doc, d.words
FROM     documents d
WHERE    '{"A", "B"}' <@ d.words
```

doc	words1
d1	$\{A, B, C\}$
d4	$\{B, B, A, D\}$
d7	$\{C, B, A\}$
d8	$\{B, A\}$

Checking for Set Containment (subset)

- “Find the pairs of different documents d1, d2 such that all words in d1 also occur as words in d2."

```
SELECT   d1.doc AS doc1, d2.doc AS doc2,
         d1.words AS words1, d2.words AS words2
FROM     documents d1, documents d2
WHERE    d1.words <@ d2.words AND
         d1.doc <> d2.doc
```

doc1	doc2	words1	words2
d1	d7	$\{A, B, C\}$	$\{C, B, A\}$
d7	d1	$\{C, B, A\}$	$\{A, B, C\}$
d8	d1	$\{B, A\}$	$\{A, B, C\}$
d8	d4	$\{B, A\}$	$\{B, B, A, D\}$
d8	d7	$\{B, A\}$	$\{C, B, A\}$

Checking for Set Equality ($A = B$)

- We may wish to check if two sets are equal
- This can again be done using the $<$@ set-containment predicate
- "Find the pairs of different documents d1, d2 that have the same words."

```
SELECT  d1.doc AS doc1, d2.doc AS doc2,
        d1.words AS words1, d2.words AS words2
FROM    documents d1, documents d2
WHERE   d1.words <@ d2.words AND
        d2.words <@ d1.words AND
        d1.doc <> d2.doc
```

doc1	doc2	words1	words2
d1	d7	$\{A, B, C\}$	$\{C, B, A\}$
d7	d1	$\{C, B, A\}$	$\{A, B, C\}$

Caveat: Do not use ARRAY equality $=$ to test set-equality

- Consider the ARRAY equality predicate '$=$'
- This predicate checks if two arrays are the same, i.e., they are equal component by component
- So '$=$' is an order-dependent predicate and should therefore not be used in our context of set predicates and operations

```
SELECT   d1.doc AS doc1, d2.doc AS doc2,
         d1.words AS words1, d2.words AS words2
FROM     documents d1, documents d2
WHERE    d1.words = d2.words AND
         d1.doc <> d2.doc
```

will return the empty set

Checking for Set Emptyness ($A = \emptyset$)

- “Find the documents that contain no words."

```
SELECT   d.doc, d.words
FROM     documents d
WHERE    d.words <@ '{}'
```

- Recall that ‘{ }’ represents the empty set

Application: Set joins

- Recall queries of the form: "Find all pairs of documents (d_1, d_2) such that **some** | **not all** | **not only** | **no** | **all** | **only** words of d_1 are in d_2."
- These set-joins can be captured using the overlap and containment predicates
- To do so, we can define polymorphic user-defined functions that stand for these set-join predicates
- We will illustrate this for the **some** (i.e., **at least one**) and **all** set joins. The other set joins can be specified in a similar fashion

Application: Set joins

- SOME (at least one) set join

```
CREATE OR REPLACE FUNCTION atLeastOne (A anyarray, B anyarray)
RETURNS boolean AS
$$
  SELECT A && B;
$$ LANGUAGE SQL;
```

- ALL set join (better called SUBSET join)
  "Is each element in *A* an element of *B*?"

```
CREATE OR REPLACE FUNCTION Each (A anyarray, B anyarray)
RETURNS boolean AS
$$
  SELECT A <@ B;
$$ LANGUAGE SQL;
```

Application: Set joins

We can then write queries with set joins as follows:

- "Find all pairs of documents (d_1, d_2) such that some words of d_1 are in d_2."

```
SELECT d1.doc, d2.doc
FROM documents d1, documents d2
WHERE atLeastOne(d1.words,d2.words)
```

- "Find all pairs of documents (d_1, d_2) such that all words of d_1 are in d_2."

  Alternatively, "Find all pairs of documents (d_1, d_2) such that d_1 only contains words that are in d_2."

```
SELECT d1.doc, d2.doc
FROM documents d1, documents d2
WHERE Each(d1.words,d2.words)
```

Determining Set Size (Cardinality) ($|A|$)

- We may wish to determine the size (cardinality) of sets
- This can be done using the ARRAY cardinality function
- “Find the number of words in each document."

```
SELECT    d.doc, cardinality(d.words) AS number_of_words
FROM      documents d
```

doc	number_of_words
d1	3
d2	3
d3	2
d4	4
d5	2
d6	3
d7	3
d8	2

Example: Queries using set cardinality

- “Find the documents with fewer than 10 words"

```
SELECT    d.doc
FROM      documents d
WHERE     cardinality(d.words) < 10
```

The UNNEST operator

- It is possible to coerce an array into a (unary) relation that contains the elements of the array
- This is done using the UNNEST operator

SELECT UNNEST(ARRAY[2,1,3,4,4]) $\rightarrow$

unnest
2
1
3
4
4

- It is possible to provide an attribute name for the elements

SELECT UNNEST(ARRAY[2,1,3,4,4]) AS A $\rightarrow$

A
2
1
3
4
4

Restructuring: the UNNEST operator

- It is possible to restructure a complex-object relation by using the UNNEST restructuring operator
- "Starting from the documents relation, create a relation of (doc, word) pairs."

```
SELECT  d.doc, UNNEST(d.words) AS word
FROM    documents d
```

→

doc	word
d1	A
d1	B
d1	C
d2	B
d2	C
d2	D
d3	A
d3	E
d4	B
d4	B
d4	A
d4	D
d5	E
d5	F
d6	A
d6	D
d6	G
d7	C
d7	B
d7	A
d8	B

Set operations: setUnion, setIntersection, and setDifference

- Using UNNEST and ARRAY construction it is also possible to define setUnion, Intersection, and Difference on sets represented as arrays
- We do this with polymorphic functions.
- Here we will show how to do this for setUnion

```
CREATE FUNCTION setUnion (A anyarray, B anyarray)  RETURNS anyarray AS
$$
SELECT ARRAY( SELECT * FROM UNNEST(A)
              UNION
              SELECT * FROM UNNEST(B));
$$ LANGUAGE SQL;
```

```
SELECT setUnion( '{1, 2, 3}'::int[], '{2, 3, 3, 5}'::int[] );
```

setUnion
{1, 2, 3, 5}

```
SELECT setUnion( '{"A", "B"}'::text[], '{"A", "C"}'::text[] );
```

setUnion
{A, B, C}

Restructuring: GROUPING (nesting)

- Reconstruder the documentWord relation
- "Restructure this relation by grouping the words of each document into a set (bag)"

documentWord

doc	word
d1	A
d1	B
d1	C
d2	B
d2	C
d2	D
d3	A
d3	E
d4	B
d4	B
d4	A
d4	D
d5	E
d5	F
d6	A
d6	D
d6	G
d7	C
d7	B
d7	A
d8	B

group words by doc
$\rightarrow$

doc	words
d1	$\{A, B, C\}$
d2	$\{B, C, D\}$
d3	$\{A, E\}$
d4	$\{B, B, A, D\}$
d5	$\{E, F\}$
d6	$\{A, D, G\}$
d7	$\{C, B, A\}$
d8	$\{B, A\}$

Restructuring: GROUPING (nesting)

- This can be done using the ARRAY constructor operation

```
SELECT   DISTINCT d.doc,
         ARRAY(SELECT d1.word
               FROM   documentWord d1
               WHERE d1.doc = d.doc) AS words
FROM     documentWord d;
```

- Notice how the parameter d is used inside the ARRAY constructor to group together the words associated with the document d
- The DISTINCT operation is essential
- This query runs in $O(|documentWord|^2)$.

Restructuring: GROUPING (nesting) using the array_agg function

- The same restructuring can also be done using the array_agg aggregate function

```
SELECT d.doc, array_agg(d.word)
FROM   documentWord d
GROUP BY (d.doc)
```

- The GROUP BY(d.doc) operation partitions the documentWord by doc values
- For each cell in this partition, the array_agg function aggregates in an array the words that are in that cell
- This query run in $O(|documentWord|)$
- So much faster than the other restructuring query

Repeated restructuring (Different views of same data)

- Starting from the documents relation, we may want to create a complex-object relation words which keeps for each word the set of documents that contain that word
- In other words, we want to do the following restructuring

doc	words
d1	$\{A, B, C\}$
d2	$\{B, C, D\}$
d3	$\{A, E\}$
d4	$\{B, B, A, D\}$
d5	$\{E, F\}$
d6	$\{A, D, G\}$
d7	$\{C, B, A\}$
d8	$\{B, A\}$

restructure:
Step 1: unnest on words;
Step 2: group docs by word
$\rightarrow$

word	docs
A	$\{d1, d3, d4, d6, d7, d8\}$
B	$\{d1, d2, d4, d7, d8\}$
C	$\{d1, d2, d7\}$
D	$\{d2, d4, d6\}$
E	$\{d3, d5\}$
F	$\{d5\}$
G	$\{d6\}$

- This can be accomplished by unnesting the documents relation on words and then grouping the doc values by word

Repeated restructuring

```
WITH docWord AS (SELECT d.doc AS doc,
                        UNNEST(d.words) AS word
                 FROM documents d)
SELECT p.word AS word, array_agg(p.doc) AS docs
FROM    docWord p
GROUP BY (p.word)
```

Or, as one query

```
SELECT      word, array_agg(doc) AS docs
FROM        (SELECT doc, UNNEST(words) AS word
             FROM    documents d) p
GROUP BY    (word)
```

Application: The word-count problem

“Determine the word-count, i.e., frequency of occurrence, of each word in the set of documents"

```
SELECT      word, cardinality(array_agg(doc)) AS wordCount
FROM        (SELECT doc, UNNEST(words) AS word
             FROM    documents d) p
GROUP BY    (word)
```

doc	words
d1	$\{A, B, C\}$
d2	$\{B, C, D\}$
d3	$\{A, E\}$
d4	$\{B, B, A, D\}$
d5	$\{E, F\}$
d6	$\{A, D, G\}$
d7	$\{C, B, A\}$
d8	$\{B, A\}$

$\rightarrow$

word	wordCount
F	1
G	1
E	2
C	3
D	3
A	6
B	6

Application: The most frequent words

"Find the words that occur most frequently in the set of documents."

```
WITH E AS (
   SELECT      word, cardinality(array_agg (doc)) AS wordCount
   FROM        (SELECT doc, UNNEST(words) AS word
                FROM     documents d) p
                GROUP BY (word))

SELECT         word
FROM           E
WHERE          wordCount = (SELECT MAX(wordCount) FROM E)
```

Double nesting

- Consider the following Enroll(sid,cno,grade) relation

sid	cno	grade
1001	2001	A
1001	2002	A
1001	2003	B
1002	2001	B
1002	2003	A
1003	2004	A
1003	2005	B
1004	2002	A
1004	2004	A
1005	2001	B
1005	2003	A

- From this we want to create a complex-object relation which stores for each student, his or her courses, internally grouped by grades obtained in these courses
- This requires double nesting

Double nesting

- We begin by grouping on (sid,grade)

```
SELECT e.sid, e.grade, array_agg(e.cno) AS courses
FROM enroll e
GROUP BY (e.sid, e.grade)
```

- This gives the complex-object relation

sid	grade	courses
1001	A	{2001, 2002}
1001	B	{2003}
1002	A	{2003}
1002	B	{2001}
1003	A	{2004}
1003	B	{2005}
1004	A	{2002, 2004}
1005	A	{2003}
1005	B	{2001}

Double nesting

- We then group over the pair of attributes (grade,courses)

```
WITH F AS (SELECT e.sid, e.grade, array_agg(e.cno) AS courses
           FROM enroll e
           GROUP BY (e.sid, e.grade))

SELECT f.sid, array_agg((f.grade, f.courses)) AS grades
FROM   F f
GROUP BY (f.sid)
```

- Notice the clause array_agg((f.grade,f.course))
- Recall that it is required to make a row (f.grade,f.course) since the array_agg function can only make an array wherein the array values are single values
- I.e., it is not allowed to write array_agg(e.grade,e.course)

Double nesting

sid	cno	grade
1001	2001	A
1001	2002	A
1001	2003	B
1002	2001	B
1002	2003	A
1003	2004	A
1003	2005	B
1004	2002	A
1004	2004	A
1005	2001	B
1005	2003	A

group by (cno)
group by(grade, courses)
$\rightarrow$

sid	grades
1001	$\{"(A, "\{2001, 2002\}")", "(B, "\{2003\})"\}$
1002	$\{"(A, "\{2003\}")", "(B, "\{2001\})"\}$
1003	$\{"(A, "\{2004\}")", "(B, "\{2005\})"\}$
1004	$\{"(A, "\{2002, 2004\}")"\}$
1005	$\{"(A, \{2003\})", "(B, \{2001\})"\}$

- For example, student 1001 obtained two types of grades: ‘A’ and ‘B’
- She received an ‘A’ in courses 2001 and 2002, and a ‘B’ is course 2003