Small. Fast. Reliable.
Choose any three.

JSON Functions And Operators

1. Overview

By default, SQLite supports fifteen functions and two operators for dealing with JSON values. There are also two table-valued functions that can be used to decompose a JSON string.

There are 15 scalar functions and operators:

1. json(*json*)
2. json_array(*value1*,*value2*,...)
3. json_array_length(*json*)
   json_array_length(*json*,*path*)
4. json_extract(*json*,*path*,...)
5. *json* -> *path*
6. *json* ->> *path*
7. json_insert(*json*,*path*,*value*,...)
8. json_object(*label1*,*value1*,...)
9. json_patch(*json*1,json2)
10. json_remove(*json*,*path*,...)
11. json_replace(*json*,*path*,*value*,...)
12. json_set(*json*,*path*,*value*,...)
13. json_type(*json*)
    json_type(*json*,*path*)
14. json_valid(*json*)
15. json_quote(*value*)

There are two aggregate SQL functions:

16. json_group_array(*value*)
17. json_group_object(name,*value*)

The two table-valued functions are:

18. json_each(*json*)
    json_each(*json*,*path*)
19. json_tree(*json*)
    json_tree(*json*,*path*)

2. Compiling in JSON Support

The JSON functions and operators are built into SQLite by default, as of SQLite version 3.38.0 (2022-02-22). They can be omitted by adding the -DSQLITE_OMIT_JSON compile-time option. Prior to version 3.38.0, the JSON functions were an extension that would only be included in builds if the -DSQLITE_ENABLE_JSON1 compile-time option was included. In other words, the JSON functions went from being opt-in with SQLite version 3.37.2 and earlier to opt-out with SQLite version 3.38.0 and later.

3. Interface Overview

SQLite stores JSON as ordinary text. Backwards compatibility constraints mean that SQLite is only able to store values that are NULL, integers, floating-point numbers, text, and BLOBs. It is not possible to add a sixth "JSON" type.

SQLite does not (currently) support a binary encoding of JSON. Experiments have been unable to find a binary encoding that is significantly smaller or faster than a plain text encoding. (The present implementation parses JSON text at over 1 GB/s.) All JSON functions currently throw an error if any of their arguments are BLOBs because BLOBs are reserved for a future enhancement in which BLOBs will store the binary encoding for JSON.

3.1. JSON arguments

For functions that accept JSON as their first argument, that argument can be a JSON object, array, number, string, or null. SQLite numeric values and NULL values are interpreted as JSON numbers and nulls, respectively. SQLite text values can be understood as JSON objects, arrays, or strings. If an SQLite text value that is not a well-formed JSON object, array, or string is passed into JSON function, that function will usually throw an error. (Exceptions to this rule are json_valid() and json_quote().)

For the purposes of determining validity, leading and trailing whitespace on JSON inputs is ignored. Interior whitespace is also ignored, in accordance with the JSON spec. These routines accept exactly the rfc-7159 JSON syntax — no more and no less.

3.2. PATH arguments

For functions that accept PATH arguments, that PATH must be well-formed or else the function will throw an error. A well-formed PATH is a text value that begins with exactly one '$' character followed by zero or more instances of ".*objectlabel*" or "[*arrayindex*]".

The *arrayindex* is usually a non-negative integer *N*. In that case, the array element selected is the *N*-th element of the array, starting with zero on the left. The *arrayindex* can also be of the form "**#-*N***" in which case the element selected is the *N*-th from the right. The last element of the array is "**#-1**". Think of the "#" characters as the "number of elements in the array". Then the expression "#-1" evaluates to the integer that corresponds to the last entry in the array. It is sometimes useful for the array index to be just the # character, for example when appending a value to an existing JSON array:

- json_set('[0,1,2]','$[#]','new') → '[0,1,2,"new"]'

3.3. VALUE arguments

For functions that accept "*value*" arguments (also shown as "*value1*" and "*value2*"), those arguments are usually understood to be literal strings that are quoted and become JSON string values in the result. Even if the input *value* strings look like well-formed JSON, they are still interpreted as literal strings in the result.

However, if a *value* argument comes directly from the result of another JSON function or from the -> operator (but not the ->> operator), then the argument is understood to be actual JSON and the complete JSON is inserted rather than a quoted string.

For example, in the following call to json_object(), the *value* argument looks like a well-formed JSON array. However, because it is just ordinary SQL text, it is interpreted as a literal string and added to the result as a quoted string:

- json_object('ex','[52,3.14159]') → '{"ex":"[52,3.14159]"}'
- json_object('ex',('52,3.14159]'->>'$')) → '{"ex":"[52,3.14159]"}'

But if the *value* argument in the outer json_object() call is the result of another JSON function like json() or json_array(), then the value is understood to be actual JSON and is inserted as such:

- json_object('ex',json('[52,3.14159]')) → '{"ex":[52,3.14159]}'
- json_object('ex',json_array(52,3.14159)) → '{"ex":[52,3.14159]}'
- json_object('ex','[52,3.14159]'->'$') → '{"ex":[52,3.14159]}'

To be clear: "*json*" arguments are always interpreted as JSON regardless of where the value for that argument comes from. But "*value*" arguments are only interpreted as JSON if those arguments come directly from another JSON function or the -> operator.

3.4. Compatibility

The current implementation of this JSON library uses a recursive descent parser. In order to avoid using excess stack space, any JSON input that has more than 2000 levels of nesting is considered invalid. Limits on nesting depth are allowed for compatible implementations of JSON by RFC-7159 section 9.

4. Function Details

The following sections provide additional detail on the operation of the various JSON functions and operators:

4.1. The json() function

The json(X) function verifies that its argument X is a valid JSON string and returns a minified version of that JSON string (with all unnecessary whitespace removed). If X is not a well-formed JSON string, then this routine throws an error.

In other words, this function converts raw text that looks like JSON into actual JSON so that it may be passed into the value argument of some other json function and will be interpreted as JSON rather than a string. This function is not appropriate for testing whether or not a particular string is well-formed JSON - use the json_valid() routine below for that task.

If the argument X to json(X) contains JSON objects with duplicate labels, then it is undefined whether or not the duplicates are preserved. The current implementation preserves duplicates. However, future enhancements to this routine may choose to silently remove duplicates.

Example:

- json(' { "this" : "is", "a": ["test"] } ') → '{"this":"is","a":["test"]}'

4.2. The json_array() function

The json_array() SQL function accepts zero or more arguments and returns a well-formed JSON array that is composed from those arguments. If any argument to json_array() is a BLOB then an error is thrown.

An argument with SQL type TEXT is normally converted into a quoted JSON string. However, if the argument is the output from another json1 function, then it is stored as JSON. This allows calls to json_array() and json_object() to be nested. The json() function can also be used to force strings to be recognized as JSON.

Examples:

- json_array(1,2,'3',4) → '[1,2,"3",4]'
- json_array('[1,2]') → '["[1,2]"]'
- json_array(json_array(1,2)) → '[[1,2]]'
- json_array(1,null,'3','[4,5]','{"six":7.7}') → '[1,null,"3","[4,5]","{\"six\":7.7}"]'
- json_array(1,null,'3',json('[4,5]'),json('{"six":7.7}')) → '[1,null,"3",[4,5],{"six":7.7}]'

4.3. The json_array_length() function

The json_array_length(X) function returns the number of elements in the JSON array X, or 0 if X is some kind of JSON value other than an array. The json_array_length(X,P) locates the array at path P within X and returns the length of that array, or 0 if path P locates an element or X other than a JSON array, and NULL if path P does not locate any element of X. Errors are thrown if either X is not well-formed JSON or if P is not a well-formed path.

Examples:

- json_array_length('[1,2,3,4]') → 4
- json_array_length('[1,2,3,4]', '$') → 4
- json_array_length('[1,2,3,4]', '$[2]') → 0
- json_array_length('{"one":[1,2,3]}') → 0
- json_array_length('{"one":[1,2,3]}', '$.one') → 3
- json_array_length('{"one":[1,2,3]}', '$.two') → NULL

4.4. The json_extract() function

The json_extract(X,P1,P2,...) extracts and returns one or more values from the well-formed JSON at X. If only a single path P1 is provided, then the SQL datatype of the result is NULL for a JSON null, INTEGER or REAL for a JSON numeric value, an INTEGER zero for a JSON false value, an INTEGER one for a JSON true value, the dequoted text for a JSON string value, and a text representation for JSON object and array values. If there are multiple path arguments (P1, P2, and so forth) then this routine returns SQLite text which is a well-formed JSON array holding the various values.

Examples:

- json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$') → '{"a":2,"c":[4,5,{"f":7}]}'
- json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.c') → '[4,5,{"f":7}]'
- json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.c[2]') → '{"f":7}'
- json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.c[2].f') → 7
- json_extract('{"a":2,"c":[4,5],"f":7}','$.c','$.a') → '[[4,5],2]'
- json_extract('{"a":2,"c":[4,5],"f":7}','$.c[#-1]') → 5
- json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.x') → NULL
- json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.x', '$.a') → '[null,2]'
- json_extract('{"a":"xyz"}', '$.a') → 'xyz'
- json_extract('{"a":null}', '$.a') → NULL

There is a subtle incompatibility between the json_extract() function in SQLite and the json_extract() function in MySQL. The MySQL version of json_extract() always returns JSON. The SQLite version of json_extract() only returns JSON if there are two or more PATH arguments (because the result is then a JSON array) or if the single PATH argument references an array or object. In SQLite, if json_extract() has only a single PATH argument and that PATH references a JSON null or a string or a numeric value, then json_extract() returns the corresponding SQL NULL, TEXT, INTEGER, or REAL value.

The difference between MySQL json_extract() and SQLite json_extract() really only stands out when accessing individual values within the JSON that are strings or NULLs. The following table demonstrates the difference:

Operation	**SQLite Result**	**MySQL Result**
json_extract('{"a":null,"b":"xyz"}','$.a')	NULL	'null'
json_extract('{"a":null,"b":"xyz"}','$.b')	'xyz'	'"xyz"'

4.5. The -> and ->> operators

Beginning with SQLite version 3.38.0 (2022-02-22), the -> and ->> operators are available for extracting subcomponents of JSON. The SQLite implementation of the -> and ->> operators strive to be compatible with both MySQL and PostgreSQL. The -> and ->> operators take a JSON string as their left operand and a PATH expression or object field label or array index as their right operand. The -> operator returns a JSON representation of the selected subcomponent or NULL if that subcomponent does not exist. The ->> operator returns an SQL TEXT, INTEGER, REAL, or NULL value that represents the selected subcomponent, or NULL if the subcomponent does not exist.

Both the -> and ->> operators select the same subcomponent of the JSON to their left. The difference is that -> always returns a JSON representation of that subcomponent and the ->> operator always returns an SQL representation of that subcomponent. Thus, these operators are subtly different from a two-argument json_extract() function call. A call to json_extract() with two arguments will return a JSON representation of the subcomponent if and only if the subcomponent is a JSON array or object, and will return an SQL representation of the subcomponent if the subcomponent is a JSON null, string, or numeric value.

The right-hand operand to the -> and ->> operators can be a well-formed JSON path expression. This is the form used by MySQL. For compatibility with PostgreSQL, the -> and ->> operators also accept a text label or integer as their right-hand operand. If the right operand is a text label X, then it is interpreted as the JSON path '$.X'. If the right operand is an integer value N, then it is interpreted as the JSON path '$[N]'.

Examples:

- '{"a":2,"c":[4,5,{"f":7}]}' -> '$' → '{"a":2,"c":[4,5,{"f":7}]}'
- '{"a":2,"c":[4,5,{"f":7}]}' -> '$.c' → '[4,5,{"f":7}]'
- '{"a":2,"c":[4,5,{"f":7}]}' -> 'c' → '[4,5,{"f":7}]'
- '{"a":2,"c":[4,5,{"f":7}]}' -> '$.c[2]' → '{"f":7}'
- '{"a":2,"c":[4,5,{"f":7}]}' -> '$.c[2].f' → '7'
- '{"a":2,"c":[4,5],"f":7}' -> '$.c[#-1]' → '5'
- '{"a":2,"c":[4,5,{"f":7}]}' -> '$.x' → NULL
- '[11,22,33,44]' -> 3 → '44'
- '[11,22,33,44]' ->> 3 → 44
- '{"a":"xyz"}' -> '$.a' → '"xyz"'
- '{"a":"xyz"}' ->> '$.a' → 'xyz'
- '{"a":null}' -> '$.a' → 'null'
- '{"a":null}' ->> '$.a' → NULL

4.6. The json_insert(), json_replace, and json_set() functions

The json_insert(), json_replace, and json_set() functions all take a single JSON value as their first argument followed by zero or more pairs of path and value arguments, and return a new JSON string formed by updating the input JSON by the path/value pairs. The functions differ only in how they deal with creating new values and overwriting preexisting values.

Function	Overwrite if already exists?	Create if does not exist?
json_insert()	No	Yes
json_replace()	Yes	No
json_set()	Yes	Yes

The json_insert(), json_replace(), and json_set() functions always take an odd number of arguments. The first argument is always the original JSON to be edited. Subsequent arguments occur in pairs with the first element of each pair being a path and the second element being the value to insert or replace or set on that path.

Edits occur sequentially from left to right. Changes caused by prior edits can affect the path search for subsequent edits.

If the value of a path/value pair is an SQLite TEXT value, then it is normally inserted as a quoted JSON string, even if the string looks like valid JSON. However, if the value is the result of another json1 function (such as json() or json_array() or json_object()) or if it is the result of the -> operator, then it is interpreted as JSON and is inserted as JSON retaining all of its substructure. Values that are the result of the ->> operator are always interpreted as TEXT and are inserted as a JSON string even if they look like valid JSON.

These routines throw an error if the first JSON argument is not well-formed or if any PATH argument is not well-formed or if any argument is a BLOB.

To append an element onto the end of an array, using json_insert() with an array index of "#". Examples:

- json_insert('[1,2,3,4]','$[#]',99) → '[1,2,3,4,99]'
- json_insert('[1,[2,3],4]','$[1][#]',99) → '[1,[2,3,99],4]'

Other examples:

- json_insert('{"a":2,"c":4}', '$.a', 99) → '{"a":2,"c":4}'
- json_insert('{"a":2,"c":4}', '$.e', 99) → '{"a":2,"c":4,"e":99}'
- json_replace('{"a":2,"c":4}', '$.a', 99) → '{"a":99,"c":4}'
- json_replace('{"a":2,"c":4}', '$.e', 99) → '{"a":2,"c":4}'
- json_set('{"a":2,"c":4}', '$.a', 99) → '{"a":99,"c":4}'
- json_set('{"a":2,"c":4}', '$.e', 99) → '{"a":2,"c":4,"e":99}'
- json_set('{"a":2,"c":4}', '$.c', '[97,96]') → '{"a":2,"c":"[97,96]"}'
- json_set('{"a":2,"c":4}', '$.c', json('[97,96]')) → '{"a":2,"c":[97,96]}'
- json_set('{"a":2,"c":4}', '$.c', json_array(97,96)) → '{"a":2,"c":[97,96]}'

4.7. The json_object() function

The json_object() SQL function accepts zero or more pairs of arguments and returns a well-formed JSON object that is composed from those arguments. The first argument of each pair is the label and the second argument of each pair is the value. If any argument to json_object() is a BLOB then an error is thrown.

The json_object() function currently allows duplicate labels without complaint, though this might change in a future enhancement.

An argument with SQL type TEXT it is normally converted into a quoted JSON string even if the input text is well-formed JSON. However, if the argument is the direct result from another JSON function or the -> operator (but not the ->> operator), then it is treated as JSON and all of its JSON type information and substructure is preserved. This allows calls to json_object() and json_array() to be nested. The json() function can also be used to force strings to be recognized as JSON.

Examples:

- json_object('a',2,'c',4) → '{"a":2,"c":4}'
- json_object('a',2,'c','{e:5}') → '{"a":2,"c":"{e:5}"}'
- json_object('a',2,'c',json_object('e',5)) → '{"a":2,"c":{"e":5}}'

4.8. The json_patch() function

The json_patch(T,P) SQL function runs the RFC-7396 MergePatch algorithm to apply patch P against input T. The patched copy of T is returned.

MergePatch can add, modify, or delete elements of a JSON Object, and so for JSON Objects, the json_patch() routine is a generalized replacement for json_set() and json_remove(). However, MergePatch treats JSON Array objects as atomic. MergePatch cannot append to an Array nor modify individual elements of an Array. It can only insert, replace, or delete the whole Array as a single unit. Hence, json_patch() is not as useful when dealing with JSON that includes Arrays, especially Arrays with lots of substructure.

Examples:

- json_patch('{"a":1,"b":2}','{"c":3,"d":4}') → '{"a":1,"b":2,"c":3,"d":4}'
- json_patch('{"a":[1,2],"b":2}','{"a":9}') → '{"a":9,"b":2}'
- json_patch('{"a":[1,2],"b":2}','{"a":null}') → '{"b":2}'
- json_patch('{"a":1,"b":2}','{"a":9,"b":null,"c":8}') → '{"a":9,"c":8}'
- json_patch('{"a":{"x":1,"y":2},"b":3}','{"a":{"y":9},"c":8}') → '{"a":{"x":1,"y":9},"b":3,"c":8}'

4.9. The json_remove() function

The json_remove(X,P,...) function takes a single JSON value as its first argument followed by zero or more path arguments. The json_remove(X,P,...) function returns a copy of the X parameter with all the elements identified by path arguments removed. Paths that select elements not found in X are silently ignored.

Removals occurs sequentially from left to right. Changes caused by prior removals can affect the path search for subsequent arguments.

If the json_remove(X) function is called with no path arguments, then it returns the input X reformatted, with excess whitespace removed.

The json_remove() function throws an error if the first argument is not well-formed JSON or if any later argument is not a well-formed path, or if any argument is a BLOB.

Examples:

- json_remove('[0,1,2,3,4]','$[2]') → '[0,1,3,4]'
- json_remove('[0,1,2,3,4]','$[2]','$[0]') → '[1,3,4]'
- json_remove('[0,1,2,3,4]','$[0]','$[2]') → '[1,2,4]'
- json_remove('[0,1,2,3,4]','$[#-1]','$[0]') → '[1,2,3]'
- json_remove('{"x":25,"y":42}') → '{"x":25,"y":42}'
- json_remove('{"x":25,"y":42}','$.z') → '{"x":25,"y":42}'
- json_remove('{"x":25,"y":42}','$.y') → '{"x":25}'
- json_remove('{"x":25,"y":42}','$') → NULL

4.10. The json_type() function

The json_type(X) function returns the "type" of the outermost element of X. The json_type(X,P) function returns the "type" of the element in X that is selected by path P. The "type" returned by json_type() is one of the following SQL text values: 'null', 'true', 'false', 'integer', 'real', 'text', 'array', or 'object'. If the path P in json_type(X,P) selects an element that does not exist in X, then this function returns NULL.

The json_type() function throws an error if any of its arguments is not well-formed or is a BLOB.

Examples:

- json_type('{"a":[2,3.5,true,false,null,"x"]}') → 'object'
- json_type('{"a":[2,3.5,true,false,null,"x"]}','$') → 'object'
- json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a') → 'array'
- json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[0]') → 'integer'
- json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[1]') → 'real'
- json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[2]') → 'true'
- json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[3]') → 'false'
- json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[4]') → 'null'
- json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[5]') → 'text'
- json_type('{"a":[2,3.5,true,false,null,"x"]}','$.a[6]') → NULL

4.11. The json_valid() function

The json_valid(X) function return 1 if the argument X is well-formed JSON and return 0 if the argument X is not well-formed JSON.

Examples:

- json_valid('{"x":35}') → 1
- json_valid('{"x":35') → 0

4.12. The json_quote() function

The json_quote(X) function converts the SQL value X (a number or a string) into its corresponding JSON representation. If X is already well-formed JSON, then this function is a no-op.

Examples:

- json_quote(3.14159) → 3.14159
- json_quote('verdant') → "verdant"
- json_quote('[1]') → '[1]'
- json_quote('[1,') → '\"[1\"'

4.13. The json_group_array() and json_group_object() aggregate SQL functions

The json_group_array(X) function is an aggregate SQL function that returns a JSON array comprised of all X values in the aggregation. Similarly, the json_group_object(NAME,VALUE) function returns a JSON object comprised of all NAME/VALUE pairs in the aggregation.

4.14. The json_each() and json_tree() table-valued functions

The json_each(X) and json_tree(X) table-valued functions walk the JSON value provided as their first argument and return one row for each element. The json_each(X) function only walks the immediate children of the top-level array or object, or just the top-level element itself if the top-level element is a primitive value. The json_tree(X) function recursively walks through the JSON substructure starting with the top-level element.

The json_each(X,P) and json_tree(X,P) functions work just like their one-argument counterparts except that they treat the element identified by path P as the top-level element.

The schema for the table returned by json_each() and json_tree() is as follows:

```
CREATE TABLE json_tree(
    key ANY,             -- key for current element relative to its parent
    value ANY,           -- value for the current element
    type TEXT,           -- 'object','array','string','integer', etc.
    atom ANY,            -- value for primitive types, null for array & object
    id INTEGER,          -- integer ID for this element
    parent INTEGER,      -- integer ID for the parent of this element
    fullkey TEXT,        -- full path describing the current element
    path TEXT,           -- path to the container of the current row
    json JSON HIDDEN,    -- 1st input parameter: the raw JSON
    root TEXT HIDDEN     -- 2nd input parameter: the PATH at which to start
);
```

The "key" column is the integer array index for elements of a JSON array and the text label for elements of a JSON object. The key column is NULL in all other cases.

The "atom" column is the SQL value corresponding to primitive elements - elements other than JSON arrays and objects. The "atom" column is NULL for a JSON array or object. The "value" column is the same as the "atom" column for primitive JSON elements but takes on the text JSON value for arrays and objects.

The "type" column is an SQL text value taken from ('null', 'true', 'false', 'integer', 'real', 'text', 'array', 'object') according to the type of the current JSON element.

The "id" column is an integer that identifies a specific JSON element within the complete JSON string. The "id" integer is an internal housekeeping number, the computation of which might change in future releases. The only guarantee is that the "id" column will be different for every row.

The "parent" column is always NULL for json_each(). For json_tree(), the "parent" column is the "id" integer for the parent of the current element, or NULL for the top-level JSON element or the element identified by the root path in the second argument.

The "fullkey" column is a text path that uniquely identifies the current row element within the original JSON string. The complete key to the true top-level element is returned even if an alternative starting point is provided by the "root" argument.

The "path" column is the path to the array or object container that holds the current row, or the path to the current row in the case where the iteration starts on a primitive type and thus only provides a single row of output.

4.14.1. Examples using json_each() and json_tree()

Suppose the table "CREATE TABLE user(name,phone)" stores zero or more phone numbers as a JSON array object in the user.phone field. To find all users who have any phone number with a 704 area code:

```
SELECT DISTINCT user.name
  FROM user, json_each(user.phone)
 WHERE json_each.value LIKE '704-%';
```

Now suppose the user.phone field contains plain text if the user has only a single phone number and a JSON array if the user has multiple phone numbers. The same question is posed: "Which users have a phone number in the 704 area code?" But now the json_each() function can only be called for those users that have two or more phone numbers since json_each() requires well-formed JSON as its first argument:

```
SELECT name FROM user WHERE phone LIKE '704-%'
UNION
SELECT user.name
  FROM user, json_each(user.phone)
 WHERE json_valid(user.phone)
   AND json_each.value LIKE '704-%';
```

Consider a different database with "CREATE TABLE big(json JSON)". To see a complete line-by-line decomposition of the data:

```
SELECT big.rowid, fullkey, value
  FROM big, json_tree(big.json)
 WHERE json_tree.type NOT IN ('object','array');
```

In the previous, the "type NOT IN ('object','array')" term of the WHERE clause suppresses containers and only lets through leaf elements. The same effect could be achieved this way:

```
SELECT big.rowid, fullkey, atom
  FROM big, json_tree(big.json)
 WHERE atom IS NOT NULL;
```

Suppose each entry in the BIG table is a JSON object with a '$.id' field that is a unique identifier and a '$.partlist' field that can be a deeply nested object. You want to find the id of every entry that contains one or more references to uuid '6fa5181e-5721-11e5-a04e-57f3d7b32808' anywhere in its '$.partlist'.

```
SELECT DISTINCT json_extract(big.json,'$.id')
  FROM big, json_tree(big.json, '$.partlist')
 WHERE json_tree.key='uuid'
   AND json_tree.value='6fa5181e-5721-11e5-a04e-57f3d7b32808';
```