When can I save JSON or XML data in an SQL Table

47

16

When using `SQL` or `MySQL` (or any relational DB for that matter) - I understand that saving the data in regular columns is better for indexing sake and other purposes...

The thing is loading and saving `JSON` data is sometimes a lot more simple. and makes the development easier.

Are there any "golden rules" for saving raw `JSON` data in the DB?

is it absolutely wrong practice to do so?

SUMMARY

Very nice answers were given, but no doubt the most well organized is the answer given by @Shnugo which deserves the bounty.

Would also like to point out answers given by @Gordon Linoff and @Amresh Pandey for explaining other special use cases.

Thank god, and good job everyone!

sql json xml normalization structured-data

edited Apr 30 '17 at 11:35
Shnugo
53k 7 28 76

asked Apr 19 '17 at 11:44
levi
1,386 1 13 33

2 I'd imagine querying on specific properties within the JSON itself could lend itself to bottlenecks. If there are specific fields that are needed for querying in the JSON, they might be candidates for extraction into their own column. Some DBs even have "json" data types, though I don't know what sort of optimizations are done using that type. – Kritner Apr 19 '17 at 11:47

8 Answers

The main questions are

- What are you going to do with this data? and

61

+50

- How are you filtering/sorting/joining/manipulating this data?

JSON (like XML) is great for data exchange, small storage and generically defined structures, but it cannot participate in typical actions you run within your RDBMS. In most cases it will be better to transfer your JSON data into *normal tables* and re-create the JSON when you need it.

XML / JSON and *1.NF*

The first rule of normalisation dictates, never to store more than one bit of information into one column. You see a column "PersonName" with a value like "Mickey Mouse"? You point to this and cry: *Change that immediately!*

What about XML or JSON? Are these types breaking 1.NF? Well, yes and no...

It is perfectly okay to store a complete structure **as one bit of information** if it is **one bit of information** actually. You get a SOAP response and want to store it because you might need this for future reference (but you will **not use this data for your own processes**)? Just store it *as is*!

Now imagine a **complex structure (XML or JSON) representing a person** (with its address, further details...). Now you put this **into one column as** `PersonInCharge` . Is this wrong? Shouldn't this rather live in properly designed related tables with a foreign key reference instead of the XML/JSON? Especially if the same person might occur in many different rows it is definitely wrong to use an XML/JSON approach.

But now imagine the need to store historical data. You want to **persist** the person's data for a given moment in time. Some days later the person tells you a new address? No problem! The old address lives in an XML/JSON if you ever need it...

Conclusion: If you store the data just to keep it, it's okay. If this data is a *unique* portion, it's okay...
But if you need the *internal parts* regularly or if this would mean redundant duplicate storage it's not okay...

Physical storage

The following is for SQL Server and might be different on other RDBMs.

XML is not stored as the text you see, but as a hierarchy tree. Querying this is astonishingly well performing! This structure is not parsed on string level!
JSON in SQL Server (2016+) lives in a string and must be parsed. There is no real native JSON type (like there is a native XML type). This might come later, but for now I'd assume, that JSON will not be as performant as XML on SQL Server (see section *UPDATE 2*). Any need to read a value out of JSON will need a hell of lot of hidden string method calls...

What does this mean for you?

your *lovable DB artist* :-D knows, that storing *JSON* ***as is***, is against common principles of RDBMs. He knows,

- that a JSON is quite probably breaking 1.NF
- that a JSON might change in time (same column, differing content).
- that a JSON is not easy to read, and it is very hard to filter/search/join or sort by it.
- that such operations will shift quite some extra load onto poor little DB server

There are some workarounds (depending on the RDBMS you are using), but most of them don't work the way you'd like it...

The answer to your question in short

YES

- If you **do not want to use data, which is stored *within* your JSON** for expensive operations (filter/join/sort).
  You can store this just as any other *exists only* content. We are storing many pictures as BLOBs, but we would not try to filter for all images with a flower...
- If you do not bother at all what's inside (just store it and read it as one bit of information)
- If the structures are variable, which would make it harder to create physical tables then to work with JSON data.
- If the structure is deeply nested, that the storage in physical tables is to much overhead

NO

- If you want to use the internal data like you'd use a relational table's data (filter, indexes, joins...)
- If you would store duplicates (create redundancy)
- In general: If you face performance problems (for sure you will face them in many typical scenarios!)

You might start with the JSON within a string column or as BLOB and change this to physical tables when you need it. My magic crystal ball tells me, this might be tomorrow :-D

UPDATE

Find some ideas about performance and disc space here: https://stackoverflow.com/a/47408528/5089204

UPDATE 2: More about performance...

The following addresses JSON and XML support in SQL-Server 2016

User @mike123 pointed to an article on an official microsoft blog which seems to proof in an experiment, that **querying a JSON is *10 x faster* then querying an XML** in SQL-Server.

Some thoughts about that:

Some cross-checks with the "experiment":

- the **"experiment" measures a lot, but not the performance of XML vs. JSON**. Doing the same action agaist the same (unchanged) string repeatedly is not a realistic scenario
- The tested examples are **far to simple for a general statement**!
- The value read is always the same and not even used. The optimizer will see this...
- Not a single word about the mighty `XQuery` support! Find a product with a given ID within an array? JSON needs to read the whole lot and use a filter afterwards using `WHERE`, while `XML` would allow an internal `XQuery predicate`. Not to speak about `FLWOR` ...
- the "experiments" code *as is* on my system brings up: JSON seems to be 3x faster (but not 10x).
- **Adding `/text()` to the `XPath` reduces this to less than 2x**. In the related article user "Mister Magoo" pointed this out already, but the *click-bait* title is still unchanged...
- With such an easy JSON as given in the "experiment" the fastest pure T-SQL approach was a combination of `SUBSTRING` and `CHARINDEX` :-D

The following code will show a more realistic experiment

- Using a JSON and an identical XML with more than one `Product` (a JSON array vs. sibling nodes)
- JSON and XML are slightly changing (10000 running numbers) and inserted into tables.
- There is an initial call agaist both tables to avoid *first-call-bias*
- All 10000 entries are read and the values retrieved are inserted to another table.
- Using `GO 10` will run through this block ten times to avoid *first-call-bias*

The final result shows clearly, that JSON is slower than XML (not that much, about 1.5x on a still very simple example).

The final statement:

- With an overly simplified example under undue circumstances JSON can be faster than XML
- Dealing with JSON is *pure string action*, while XML is parsed and transformed. This is rather expensive in the first action, but will speed up everything, once this is done.
- JSON might be better in a *one-time* action (avoids the overhead of creating an internal hierarchical representation of an XML)

- With a still very simple but more realistic example XML will be faster in simple reading
- Whenever there is any need to read a specific element out of an array, to filter all entries where a given ProductID is included in the array, or to navigate up and down the path, JSON cannot hold up. It must be parsed out of a string completely - each time you have to grab into it...

The test code

```
USE master;
GO
--create a clean database
CREATE DATABASE TestJsonXml;
GO
USE TestJsonXml;
GO
--create tables
CREATE TABLE TestTbl1(ID INT IDENTITY,SomeXml XML);
CREATE TABLE TestTbl2(ID INT IDENTITY,SomeJson NVARCHAR(MAX));
CREATE TABLE Target1(SomeString NVARCHAR(MAX));
CREATE TABLE Target2(SomeString NVARCHAR(MAX));
CREATE TABLE Times(Test VARCHAR(10),Diff INT)
GO
--insert 10000 XMLs into TestTbl1
WITH Tally AS(SELECT TOP 10000 ROW_NUMBER() OVER(ORDER BY (SELECT NULL))*2 AS Nmbr FROM
master..spt_values AS v1 CROSS APPLY master..spt_values AS v2)
INSERT INTO TestTbl1(SomeXml)
SELECT
N'<Root>
    <Products>
    <ProductDescription>
        <Features>
            <Maintenance>' + CAST(Nmbr AS NVARCHAR(10)) + ' year parts and labor
extended maintenance is available</Maintenance>
            <Warranty>1 year parts and labor</Warranty>
        </Features>
        <ProductID>' + CAST(Nmbr AS NVARCHAR(10)) + '</ProductID>
        <ProductName>Road Bike</ProductName>
    </ProductDescription>
    <ProductDescription>
        <Features>
            <Maintenance>' + CAST(Nmbr + 1 AS NVARCHAR(10)) + ' blah</Maintenance>
            <Warranty>1 year parts and labor</Warranty>
        </Features>
        <ProductID>' + CAST(Nmbr + 1 AS NVARCHAR(10)) + '</ProductID>
        <ProductName>Cross Bike</ProductName>
    </ProductDescription>
    </Products>
```

```
</Root>'
FROM Tally;

--insert 10000 JSONs into TestTbl2
WITH Tally AS(SELECT TOP 10000 ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS Nmbr FROM
master..spt_values AS v1 CROSS APPLY master..spt_values AS v2)
INSERT INTO TestTbl2(SomeJson)
SELECT
N'{
    "Root": {
        "Products": {
            "ProductDescription": [
                {
                    "Features": {
                        "Maintenance": "' + CAST(Nmbr AS NVARCHAR(10)) + ' year parts
and labor extended maintenance is available",
                        "Warranty": "1 year parts and labor"
                    },
                    "ProductID": "' + CAST(Nmbr AS NVARCHAR(10)) + '",
                    "ProductName": "Road Bike"
                },
                {
                    "Features": {
                        "Maintenance": "' + CAST(Nmbr + 1 AS NVARCHAR(10)) + ' blah",
                        "Warranty": "1 year parts and labor"
                    },
                    "ProductID": "' + CAST(Nmbr + 1 AS NVARCHAR(10)) + '",
                    "ProductName": "Cross Bike"
                }
            ]
        }
    }
}'
FROM Tally;
GO

--Do some initial action to avoid first-call-bias
INSERT INTO Target1(SomeString)
SELECT SomeXml.value('(/Root/Products/ProductDescription/Features/Maintenance/text())
[1]', 'nvarchar(4000)')
FROM TestTbl1;
INSERT INTO Target2(SomeString)
SELECT JSON_VALUE(SomeJson,
N'$.Root.Products.ProductDescription[0].Features.Maintenance')
FROM TestTbl2;
GO

--Start the test
DECLARE @StartDt DATETIME2(7), @EndXml DATETIME2(7), @EndJson DATETIME2(7);
```

```
--Read all ProductNames of the second product and insert them to Target1
SET @StartDt = SYSDATETIME();
INSERT INTO Target1(SomeString)
SELECT SomeXml.value('(/Root/Products/ProductDescription/ProductName/text())[2]',
'nvarchar(4000)')
FROM TestTbl1
ORDER BY NEWID();
--remember the time spent
INSERT INTO Times(Test,Diff)
SELECT 'xml',DATEDIFF(millisecond,@StartDt,SYSDATETIME());

--Same with JSON into Target2
SET @StartDt = SYSDATETIME();
INSERT INTO Target2(SomeString)
SELECT JSON_VALUE(SomeJson, N'$.Root.Products.ProductDescription[1].ProductName')
FROM TestTbl2
ORDER BY NEWID();
--remember the time spent
INSERT INTO Times(Test,Diff)
SELECT 'json',DATEDIFF(millisecond,@StartDt,SYSDATETIME());

GO 10 --do the block above 10 times

--Show the result
SELECT Test,SUM(Diff) AS SumTime, COUNT(Diff) AS CountTime
FROM Times
GROUP BY Test;
GO
--clean up
USE master;
GO
DROP DATABASE TestJsonXml;
GO
```

The result (SQL Server 2016 Express on an Acer Aspire v17 Nitro Intel i7, 8GB Ram)

```
Test    SumTime
------------------
json    2706
xml     1604
```

edited Sep 26 '18 at 6:53

answered Apr 19 '17 at 11:53

Shnugo

53k 7 28 76

This article suggests json performs x10 better than xml blogs.msdn.microsoft.com/sqlserverstorageengine/2017/11/13/… – mike123 Jan 30 '18 at 18:45

1 @mike123, read the section UPDATE 2... – Shnugo Feb 1 '18 at 9:41

Could you put nanoseconds instead of milliseconds in DATEDIFF? – Jovan MSFT Feb 8 '18 at 8:07

@JovanMSFT Sure: json: 2281502100 and xml:1296990300. That means, that XML is almost twice as fast... – Shnugo Feb 8 '18 at 8:15

What version are you using 2016, 2017? On SQL 2017 Express, I'm getting close numbers: json 1918864000 xml 1807237200 – Jovan MSFT Feb 8 '18 at 21:35

0

PostgreSQL has a built-in `json` and `jsonb` data type

- json
- json vs jsonb

These are a few examples:

```
CREATE TABLE orders (
 ID serial NOT NULL PRIMARY KEY,
 info json NOT NULL
);

INSERT INTO orders (info)
VALUES
 (
 '{ "customer": "Lily Bush", "items": {"product": "Diaper","qty": 24}}'
 ),
 (
 '{ "customer": "Josh William", "items": {"product": "Toy Car","qty": 1}}'
 ),
 (
 '{ "customer": "Mary Clark", "items": {"product": "Toy Train","qty": 2}}'
 );
```

PostgreSQL provides two native operators `->` and `->>` to query JSON data.

The operator `->` returns JSON object field by key.

The operator `->>` returns JSON object field by text.

```
SELECT
 info -> 'customer' AS customer
FROM
 orders;

SELECT
 info ->> 'customer' AS customer
FROM
 orders
WHERE
 info -> 'items' ->> 'product' = 'Diaper'
```

answered Mar 9 at 16:52

Galina Alperovich
607 7 12

8

New SQL Server provides functions for processing JSON text. Information formatted as JSON can be stored as text in standard SQL Server columns and SQL Server provides functions that can retrieve values from these JSON objects.

```
    DROP TABLE IF EXISTS Person

CREATE TABLE Person
( _id int identity constraint PK_JSON_ID primary key,
value nvarchar(max)
CONSTRAINT [Content should be formatted as JSON]
CHECK ( ISJSON(value)>0 )
)
```

This simple structure is similar to the standard NoSQL collection that you can create in NoSQL databases (e.g. Azure DocumentDB or MongoDB) where you just have key that represents ID and value that represents JSON.

Note that NVARCHAR is not just a plain text. SQL Server has built-in text compressions mechanism that can transparently compress data stored on disk. Compression depends on language and can go up to 50% depending on your data (see UNICODE compression).

The key difference between SQL server and other plain NoSQL databases is that SQL Server enables you to use hybrid data model where you can store several JSON objects in the same “collection” and combine them with regular relational columns.

As an example, imagine that we know that every person in your collection will have FirstName and LastName, and that you can store general information about the person as one JSON object, and phone numbers/email addresses as separate objects. In SQL Server 2016 we can easily create this structure without any additional syntax:

```
DROP TABLE IF EXISTS Person

CREATE TABLE Person (

 PersonID int IDENTITY PRIMARY KEY,

 FirstName nvarchar(100) NOT NULL,

 LastName nvarchar(100) NOT NULL,

 AdditionalInfo nvarchar(max) NULL,

 PhoneNumbers nvarchar(max) NULL,

 EmailAddresses nvarchar(max) NULL
 CONSTRAINT [Email addresses must be formatted as JSON array]
 CHECK ( ISJSON(EmailAddresses)>0 )

 )
```

Instead of single JSON object you can organize your data in this “collection”. If you do not want to explicitly check structure of each JSON column, you don’t need to add JSON check constraint on every column (in this example I have added CHECK constraint only on EmailAddresses column).

If you compare this structure to the standard NoSQL collection, you might notice that you will have faster access to strongly typed data (FirstName and LastName). Therefore, this solution is good choice for hybrid models where you can identify some information that are repeated across all objects, and other variable information can be stored as JSON. This way, you can combine flexibility and performance.

If you compare this structure with the schema of Person table AdventureWorks database, you might notice that we have removed many related tables.

Beside simplicity of schema, your data access operations will be simpler compared to complex relational structure. Now you can read single table instead of joining several tables. When you need to insert new person with related information (email addresses, phone numbers) you can insert a single record in one table instead of inserting one record in AdventureWorks Person table, taking identity column to find foreign key that will be used to store phones, email addresses, etc. In addition, in this model you can easily delete single person row without cascade deletes using foreign key relationships.

NoSQL databases are optimized for simple, read, insert, and delete operations – SQL Server 2016 enables you to apply the same logic in relational database.

JSON constraints In the previous examples, we have seen how to add simple constraint that validates that text stored in the column is properly formatted. Although JSON do not have strong schema, you can also add complex constraints by combining functions that read

values from JSON and standard T-SQL functions:

```
ALTER TABLE Person
 ADD CONSTRAINT [Age should be number]
 CHECK ( ISNUMERIC(JSON_VALUE(value, '$.age'))>0 )

 ALTER TABLE Person
 ADD CONSTRAINT [Person should have skills]
 CHECK ( JSON_QUERY(value, '$.skills') IS NOT NULL)
First constraint will take the value of $.age property and check is this numeric value.
Second constraint will try to find JSON object in $.skills property and verify that it
exists. The following INSERT statements will fail due to the violation of constraints:



INSERT INTO Person(value)
 VALUES ('{"age": "not a number", "skills":[]}')

 INSERT INTO Person(value)
 VALUES ('{"age": 35}')
```

Note that CHECK constraints might slow down your insert/update processes so you might avoid them if you need faster write performance.

Compressed JSON storage If you have large JSON text you can explicitly compress JSON text using built-in COMPRESS function. In the following example compressed JSON content is stored as binary data, and we have computed column that decompress JSON as original text using DECOMPRESS function:

```
CREATE TABLE Person

 ( _id int identity constraint PK_JSON_ID primary key,

 data varbinary(max),

 value AS CAST(DECOMPRESS(data) AS nvarchar(max))

 )



 INSERT INTO Person(data)

 VALUES (COMPRESS(@json))
```

COMPRESS and DECOMPRESS functions use standard GZip compression. If your client can handle GZip compression (e.g browser that understands gzip content), you can directly return compressed content. Note that this is performance/storage trade-off. If you frequently query compressed data you mig have slower performance because text must be decompressed each time.

Note: JSON functions are available only in SQL Server 2016+ and Azure SQL Database.

More can be read from the source of this article

https://blogs.msdn.microsoft.com/sqlserverstorageengine/2015/11/23/storing-json-in-sql-server/

edited Apr 28 '17 at 10:59

answered Apr 28 '17 at 7:31

AMRESH PANDEY

152 8

2

Json's are not great in relationional db's. If you unfold the json into columns and store in a db , it's great but storing a json as a blob is next to using it as data archival system.

There could be several reasons for not unfolding a json and storing it in a single column but the decision would have been taken as the values in that json field would not be used for any querying (or the values have been already unfolded into columns).

Also , most of the json processing if at all the field was queried would be outside the sql environment as sql is just not meant for json processing. The real question then becomes , where do i store this json, do i just let it be as flat files and when required query them via some other system (spark/hive/etc).

I would agree with your DB artist , don't use RDBMS for archival. There are cheaper options. Also json blobs can get huge and can start bogging down the DB disk space with time.

answered Apr 27 '17 at 7:34

Satyadev

483 3 10

4

The question you have to ask is:

> Am I tied to using only this database?

DO

1. If you can use a different database to store JSON, use a document storage solution such as CouchDB, DynamoDB or MongoDB.
2. Use these document storage DB's ability to index and search hierarchical data.
3. Use a relational database for your relational data.
4. Use a relational database for reporting, data warehousing and data mining.

DON'T

1. Store JSON as string if possible.
2. Try and come up with max length of JSON data.
3. Use varchar to store JSON (use text/blob if you must).
4. Try and search through stored JSON for values.
5. Worry about escaping JSON to store as string.

answered Apr 26 '17 at 20:48

Anand
1,083 8 17

This is the formation I was looking for, clean and simple. but it does not contain some of the important issues some of the other answers address. if other cases could be added to this, would be great – levi Apr 27 '17 at 20:43

4

The "golden rule" I use, in a hand-wavey sort of way, is that if I need JSON in its raw format, it's okay to store. If I have to make a special point of parsing it, then it's not.

For instance, if I'm creating an API that sends out raw JSON, and for whatever reason this value isn't going to change, then it's *okay* to store it as raw JSON. If I have to parse it, change it, update it, etc... then not so much.

answered Apr 25 '17 at 13:44

piisexactly3
583 3 11

I'll wave my magic wand. Poof! Golden Rules on use of JSON:

10

- If MySQL does not need to look *inside* the JSON, and the application simply needs a collection of stuff, then JSON is fine, possibly even better.
- If you will be searching on data that is inside *and* you have MariaDB 10.0.1 or MySQL 5.7 (with a JSON datatype and functions), then JSON *might* be practical. MariaDB 5.3's "Dynamic" columns is a variant on this.
- If you are doing "Entity-Attribute-Value" stuff, then JSON is not good, but it is the least of several evils. http://mysql.rjweb.org/doc.php/eav
- For searching by an indexed column, not having the value buried inside JSON is a big plus.
- For searching by a range on an indexed column, or a `FULLTEXT` search or `SPATIAL`, JSON is not possible.
- For `WHERE a=1 AND b=2` the "composite" index `INDEX(a,b)` is great; probably can't come close with JSON.
- JSON works well with "sparse" data; INDEXing works, but not as well, with such. (I am referring to values that are 'missing' or NULL for many of the rows.)
- JSON can give you "arrays" and "trees" without resorting to extra table(s). But dig into such arrays/trees *only* in the app, *not* in SQL.
- JSON is worlds better than XML. (My opinion)
- If you do not want to get into the JSON string except from the app, then I recommend compressing (in the client) it an storing into a `BLOB`. Think of it like a .jpg -- there's stuff in there, but SQL does not care.

State your application; maybe we can be more specific.

answered Apr 24 '17 at 0:19

Rick James
76.1k 5 70 112

The bullets are good, if you could make a distinct "when to" and "when not" could make this even better – levi Apr 27 '17 at 20:34

1 @levi - Yes, but several are not absolutely to/not; rather they depend on details in the situation. – Rick James Apr 28 '17 at 1:15

2 *JSON is worlds better than XML. (My opinion)* Well, JSON is less characters... What can you do with JSON, what you cannot do with XML? The most important part is: **How is this type treated?** Parsing XML or JSON with string methods will be a pain in the neck. Transforming the structure into an object tree, will allow much better approachs. SQL Server stores XML in a tree natively, but JSON will - AFAIK - live in a string... Why do you prefer JSON as **worlds better**? – Shnugo Apr 28 '17 at 7:41

2 @Shnugo - Easier to read, shorter, essentially one, *unambiguous* way to represent an array. (XML have several, most of which can be abused by duplicating the key, or whatever.) Ditto for Hash. This makes mapping to/from most computer languages straightforward. (Yes, this is my "opinion".) – Rick James Apr 28 '17 at 18:43

@RickJames The way I like to put it is that "JSON has no class" (in the sense of classes in a programming language) - it is great for completely generic lists and hashes, but immediately becomes more complex if you want to define specific custom data structures. In the context of a DB, XML (if

supported) would obviously be better if the input is XML, rather than somehow converting it to JSON (people do it, and the results are never pretty). – IMSoP May 25 '18 at 11:19

11

This is too long for a comment.

If it were "absolutely wrong", then most databases would not support it. Okay, most databases support commas in the `FROM` clause and I view that as "absolutely wrong". But support for JSON is new development, not a backward-compatible "feature".

One obvious case is when the JSON struct is simply a BLOB that is passed back to the application. Then there is no debate -- other then the overhead of storing JSON, which is unnecessarily verbose for structured data with common fields in every record.

Another case is the "sparse" columns case. You have rows with many possible columns, but these vary from row to row.

Another case is when you want to store "nested" records in a record. JSON is powerful.

If the JSON has common fields across records that you want to query on, then you are usually better off putting these in proper database columns. However, data is complicated and there is a place for formats such as JSON.

answered Apr 19 '17 at 11:51

Gordon Linoff
830k 38 341 446