Database - Feature #6418

represent extent fields as arrays

05/26/2022 03:23 PM - Eric Faulhaber

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Radu Apetrii	**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

Related issues:

Related to Database - Feature #6348: implement support for MariaDB	**WIP**
Related to Database - Support #4060: investigate converting extent fields to ...	**New**
Related to Database - Feature #6898: PostgreSQL schema improvements	**New**

History

#1 - 05/26/2022 03:49 PM - Eric Faulhaber

Our current support for extent fields is to

- (default) normalize each set of extent fields of the same extent *N* in a table into a secondary table with *N* rows for each row in the primary table; or
- (optional) denormalize an extent field of size *N* into *N* uniquely named columns in the primary table.

Neither option is ideal.

Normalization creates a lot of rows and uses a lot of storage. Even if most extent values are empty, *N* rows are created for each row in the primary table. Record retrieval is complicated and slower than if a single, simple query could be issued to get all data for a record in the primary table, including the extent data.

Denormalization is messy and can create very wide tables, as extent fields are exploded into many columns. Record retrieval requires a single, fairly straightforward query, but hydrating a DMO is more complicated. Some 4GL code which references extent field elements in queries can complicate conversion and runtime access.

If we had native array support for all data types in all databases, this might be a good alternative. Alas, support for arrays as a column data type varies wildly among databases. PostgreSQL seems to have pretty straightforward support for array data types on the face of it. You declare them as a column data type the way one would expect and they offer array functions to deal with them. I am curious about performance, for instance when when filtering a query on array content.

For the other databases we currently support or want to support in the near term, array support seems quite messy, as these vendors seem to leave arrays as a column data type up to the user to implement :(

AFAICT, MariaDB offers no native array data type which can be assigned to a column. The research I have done so far suggests using the JSON type to mimic an array. Does this mean everything is stored as a JSON string? I need to understand how the JSON data type works more clearly to understand the implications of this.

SQL Server also offers no native array data type for columns. So far, I've only found suggestions to declare a "table variable", which in this use case essentially seems to be an alias for a two column table acting as a single-dimensional array. This seems essentially to be normalization behind some smoke and mirrors, but maybe it makes accessing the array data more natural? Not sure.

H2 requires some research, though I would expect some support, since Java natively supports arrays. But I haven't looked yet. Other future candidates (e.g., Oracle, DB2/UDB, etc.) need to be assessed as well. I would guess from Oracle's kitchen-sinkness that arrays are supported, but that's only a guess.

Based on my preliminary research, I'm not too encouraged on the idea...

#2 - 05/26/2022 03:52 PM - Greg Shah

If it represents a significant improvement (in performance or at least in the SQL that accesses these tables), then we may still want to offer this on those databases which allow it.

#3 - 07/19/2022 01:57 PM - Eric Faulhaber

- Related to Feature #6348: implement support for MariaDB added

#4 - 07/19/2022 01:57 PM - Eric Faulhaber

Eric Faulhaber wrote:

> AFAICT, MariaDB offers no native array data type which can be assigned to a column. The research I have done so far suggests using the JSON type to mimic an array. Does this mean everything is stored as a JSON string? I need to understand how the JSON data type works more clearly to understand the implications of this.

A useful article on this topic: https://mariadb.com/resources/blog/using-json-in-mariadb/

#5 - 07/19/2022 03:20 PM - Greg Shah

Downsides of the JSON approach:

- The syntax is pretty messy since you have to use functions to access/set elements in the JSON approach, whereas in a denormalized case you are dealing directly with a column. Every 4GL access/set will have to be rewritten at runtime to use these functions. Any direct JDBC access will be similarly messy.
- Potentially slow. There is JSON parsing with every access and parsing/modification/anti-parsing for every set. This happens per row. And there is JSON validation happening at some unknown times (probably with the set or at least with an insert). This has got to be relatively costly.
- The highly structured nature of the original types is lost and now any kind of structures can be embedded inside this column. This may make the column much slower if a programmer decides to be clever and store random junk. Or it might break the implicit structure assumed by the array implementation in FWD, causing unknown downstream breakage at runtime. At a minimum the intention of the highly structured nature of these fields is hidden. Yuck.

I don't see much upside. Maybe for large extents the storage is slightly better, maybe. It is not a clear advantage. The disadvantages seem large in comparison. This JSON approach does not seem like an improvement over denormalized fields. I'm especially worried about the performance characteristics of all this overhead added.

#6 - 09/02/2022 09:07 AM - Alexandru Lungu

- Assignee set to Radu Apetrii

- Status changed from New to WIP

#7 - 09/05/2022 02:58 AM - Radu Apetrii

- File table_normalized_vs_denormalized.png added

I have tested these two methods of representing extent fields (normalized and denormalized) over a series of tests and here are the results:

	Create	Update	Read	Delete	Total	Size
Extent 2 normalized	**311**	**2344.75**	**4137**	**285**	**7077.75**	**704 KB**
Extent 2 denormalized	**202**	**1190.5**	**1664**	**187.5**	**3244**	**280 KB**
Extent 5 normalized	**420.5**	**5397.5**	**13766.5**	**287.5**	**19872**	**2048 KB**
Extent 5 denormalized	**210**	**1189.25**	**3183**	**195**	**4777.25**	**320 KB**
Extent 10 normalized	**584.25**	**14919**	**7249.5**	**340**	**23092.75**	**6152 KB**
Extent 10 denormalized	**215**	**1308.5**	**2978.5**	**205.25**	**4707.25**	**440 KB**
Extent 100 normalized	**2682**	**26743.5**	**12180.5**	**436.25**	**42042.25**	**16673 KB**
Extent 100 denormalized	**285.75**	**1545.75**	**1703.75**	**192.75**	**3728**	**1808 KB**

There were 4 tables used, each one containing an extent field and a column with an index. The extent field had a size of 2 for the first table, a size of 5 for the second one, a size of 10 for the third one and a size of 100 for the last one.
The tests contained:

- Create: 1000 create statements for each table.
- Update: A mix of update statements, using WHERE clause, BY clause, both of them or none of them, 1000 of each one.
- Read: A mix of read statements, using join, outer join, USE-INDEX keyword or none of them, 1000 of each one.
- Delete: 1000 delete statements for each table, leaving it empty.

The tests were run a couple of times, thus, the numbers from the table represent the average results. Time was calculated by using the converted etime method. Size of each table was obtained by using a Select statement inside PostgreSQL.
The question I would like to ask is: Is there any reason why normalized is preferred over denormalized, since the latter one provides better results regarding both time and memory used?
Also, I will start testing the PostgreSQL support for array data types next.

#8 - 09/05/2022 04:28 AM - Greg Shah

> The question I would like to ask is: Is there any reason why normalized is preferred over denormalized, since the latter one provides better results regarding both time and memory used?

Speaking for myself, I would like to brutally murder all normalized extents. The normalized approach is very unclean from a schema and coding perspective. So it is unwanted in general. If we cannot find cases where they are faster, then I think that support should be completely removed.

#9 - 09/07/2022 09:43 AM - Radu Apetrii

- File PostgreSQL_data_types_table.png added

I've done some more testing, this time regarding the SQLs for three different approaches of storing an extent field as an array.

	Create	Update	Update (bulked)	Read	Delete	Total	Total (bulked)	Total Size (KB)
PostgreSQL Array	**523.75**	**3106**	**2347.75**	**7549.25**	**446.5**	**11625.5**	**10867.25**	**3464**
Table denormalized	**708**	**3279.25**	**-**	**7761.25**	**510.75**	**12259.25**	**-**	**2848**
PostgreSQL JSON	**717.75**	**6530**	**2857.25**	**8277.5**	**485.75**	**16011**	**12338.25**	**6424**

The tables used were the same as in the previous post, the only difference between them being the data type of the extent column. The first test (PostgreSQL Array) had a column of type integer[], the second one (Table denormalized) consisted of the denormalized table (described previously) and the last one (PostgreSQL JSON) had a column of type jsonb (which is similar to json, but with data being stored in binary mode and with some characteristic functions for data manipulation). Time was measured with Apache Jmeter and size was obtained by using a Select statement inside PostgreSQL. The tests were the same as before, but now, only the time of the SQLs was measured, not of the whole running process.
Also, as seen in the table, there is a difference between Update and Update (bulked). When the field that contains the extent needs to be fully updated, the Update test updates every "sub-field" one at a time, while Update (bulked) updates the whole field in one go.
Some notes that can be taken into account are:

- Storing data as JSON is a method that decreases readability, as it often produces long lines of code, nor does it provide encouraging results. I don't think it would be a wise decision to use JSONs for this in the future.
- Arrays in PostgreSQL seem to be working well, but maybe not as well as expected. They do provide a slightly better result than the denormalized method, but I'm not entirely sure it is enough in order to be worth implementing this idea from scratch. Perhaps I'll vary the tests a bit to see how much better the arrays really are.

Should I run these tests for MariaDB and/or H2 (for temporary tables) as well?

#10 - 09/09/2022 06:05 AM - Radu Apetrii

Small update on this topic: I ran the tests from the previous post, but with a small change regarding the tables. Instead of storing extent fields of type integer, this time they stored information of type character (or text, as it appears in PostgreSQL), and the tests were updated accordingly. I won't upload the results because they are very similar to the ones before and do not provide additional notes on the matter. Basically, the denormalized method seems to be the most viable one (as I'm writing this) and based on this fact I will:

- Test this method thoroughly to make sure that it works on a large set of tests and does not provide inaccurate results.
- Take a closer look upon the custom-extent tag as I remember it did not work on all cases.
- Try to create a new tag which should denormalize all extents of all tables (and of course test it).

I will provide updates as soon as I get results.

#11 - 09/16/2022 05:16 AM - Radu Apetrii

A few updates:

- I've tested the correctness of results regarding denormalized extent fields. Out of 108 tests that run in 4GL, 72 of them give the correct result with the normalized method and 75 of them give the correct result with the denormalized method, which is a really good sign. The tests were taken from gcd/testcases/uast, more exactly, they represented a modified version (in order to include extent fields) of adaptive_scrolling, buffer and query folders that contained the files. All in all, out of the two methods, definitely denormalized is the one preferred and I think it should become the default method.
- As an extent of the last point, I believe that the easiest way of achieving this is to configure the cfg/p2j.cfg.xml file by adding the tag that Eric created (#6561): <parameter name="denorm-extents" value="true"/>, instead of modifying the hints file. IMO, the latter one should be used only if the customer wants to personalize the labels of the extent columns. Thus, I think the cfg/p2j.cfg.xml file should have the denorm-extents tag by default.
- You can correct me if I'm wrong, but I assume that the initial intention of the custom-extent tag inside the hints file was to denormalize all extent fields of a table, if there were none specified. Upon applying this tag and running the conversion, only the first extent field of the table would become denormalized, while the others would remain normalized. Therefore, I made a slight change inside the getCustomFieldExtent function in TableHints.java in order to denormalize all extent fields of a table at conversion. I will commit this change later on today if it is ok.
- There is still a debate about the approach that PostgreSQL proposes (arrays), whether it is worth taking the time in order to implement it in P2J. Based on previous results, there is one significant improvement and it is in the reading department, but only for fields that contain large extents (~100). Other than that, denormalized extents provide very similar results to the PostgreSQL arrays and much better results than PostgreSQL JSONs.

I am waiting for your decision on the last point due to the fact that there are more than two solutions that I can think about in this moment: don't integrate PostgreSQL arrays in P2J at all, fully integrate them in P2J or integrate them, but make them be used only for large extents (~100).

#12 - 09/16/2022 08:36 AM - Greg Shah

Some things to note:

- Customers hate normalized mode. We should simplify the runtime and get rid of it since there is no advantage. This will make the code easier to support and more stable (all things being equal).
- Customers would prefer the PostgreSQL array syntax (subscripting) to the denormalized mode. If there is no disadvantage to using it, then I think it is best to use it for PostgreSQL databases. The issue here is that we need to implement database independent code in the converted Java. We can implement everything as the array syntax and rewrite the subscripts in the runtime for databases that don't support the syntax directly. I prefer this approach. It matches the original syntax and makes the code closer to the original intent so I think it is better from that perspective too.
- For databases that only support JSON, we should use the denormalized approach. I don't agree with using JSON for those customers that are willing to take the completely unnecessary performance hit + nasty syntax hit. We have no business implementing the JSON concept. It is terrible in every way and makes no sense that I can see. It only serves to make our runtime more complicated (harder to support, more fragile and more error prone) and I don't see why any customer would want it.

In short: denormalized for databases that can't do array syntax, array syntax for databases that support it, no JSON and no normalized.

#13 - 09/16/2022 08:36 AM - Greg Shah

The above is my opinion. Eric makes the final decision here.

#14 - 10/20/2022 09:13 AM - Radu Apetrii

Greg Shah wrote:

> The above is my opinion. Eric makes the final decision here.

Any decision yet?

#15 - 11/01/2022 08:30 AM - Greg Shah

- Related to Support #4060: investigate converting extent fields to array columns added

#16 - 11/01/2022 08:35 AM - Greg Shah

- Related to Feature #6898: PostgreSQL schema improvements added

#17 - 11/01/2022 11:11 AM - Eric Faulhaber

Radu Apetrii wrote:

> Greg Shah wrote:
>
> > The above is my opinion. Eric makes the final decision here.
>
> Any decision yet?

I agree with Greg that we want to be rid of the "normalized" approach. If there is good array support in a database (like PostgreSQL) and we don't see any serious performance penalty vs. the denormalized approach, we should take advantage of the array support. That being said, we will need wider testing with real applications to be able to compare denormalized vs. native array performance to make the final call on the default choice. For any given database, if the array support is not good (functionally or performance-wise), we fall back to denormalized as the default for that dialect. Normalized is right out.

#18 - 11/17/2022 03:29 AM - Alexandru Lungu

Eric, is there any prior partial implementation/discussion into the array fields? If no, this task can be the lead effort into the array field implementation.

Maybe a 6418a branch should be used for this.

#19 - 11/17/2022 12:02 PM - Eric Faulhaber

Alexandru Lungu wrote:

> Eric, is there any prior partial implementation/discussion into the array fields?

There is not.

> If no, this task can be the lead effort into the array field implementation. Maybe a 6418a branch should be used for this.

Makes sense. Please base this branch off the existing 6129b branch.

Please note that we've decided not to use the JSON Array approach, nor any similar approach which implements arrays via some sort of packing/unpacking paradigm, requiring unnatural SQL syntax (e.g., get/set functions, type conversion, etc.) to access the array elements.

Also, consider the scope of this task to be broader now than just "use arrays for extent fields". The goal is to use arrays where appropriate (currently only PostgreSQL, AFAIK), and to use the denormalized/expanded approach in all other cases. We want to deprecate the use of the normalized approach altogether.

The tricky part of this is that currently, the choice of normalized vs. denormalized actually changes the converted code. For normalized, the converted code still uses array-like syntax, whereas for denormalized, it references (in most cases) discrete fields for the individual extent field elements. I never liked that we had this distinction. While it makes some sense for the code to reflect the structure of the schema, the denormalized schema is not structured that way because of the original database designer's intent; that intent was to use an array. Rather, it is an implementation detail of mapping an OE schema to a new RDBMS.

Ideally, I think the converted code should use the array-like syntax (which is closest to the original code's idiom), regardless of the choice of back-end. This is especially important since we intend to support native arrays in some database dialects and not others. If the code requires re-conversion to switch to a different database dialect, we lose a lot of deployment flexibility.

As I note above, this already is a problem with our current choice between normalized and denormalized extent fields. I think it is time to resolve this.

Files

table_normalized_vs_denormalized.png	41.1 KB	09/05/2022	Radu Apetrii
PostgreSQL_data_types_table.png	18.7 KB	09/07/2022	Radu Apetrii