

Database - Bug #11956

improvements to FTS migration

10/08/2026 07:46 AM - Greg Shah

Status:	WIP	Start date:	
Priority:	Urgent	Due date:	
Assignee:	Stefanel Pezamosca	% Done:	10%
Category:		Estimated time:	0.00 hour
Target version:			
billable:	No	reviewer:	Ovidiu Maxiniuc
vendor_id:	GCD	production:	No
case_num:		env_name:	
version_reported:		topics:	
version_resolved:			
Description			

History

#1 - 10/08/2026 07:56 AM - Greg Shah

- Priority changed from Normal to Urgent

The migration needs to handle the problem of location of the computed tsvector columns. They should not be at the end, but need to be glued (adjacent) to the column(s) on which the word index is applied.

As far as I know, the core implementation needs to change as well, so this is urgent to get done before customers migrate their production databases.

#4 - 10/08/2026 08:01 AM - Constantin Asofiei

Stefanel, another question: if there will be a need to rebuild the tsvector columns, this should be able to be done without dropping and adding them again, right?

#5 - 10/08/2026 08:13 AM - Stefanel Pezamosca

Constantin Asofiei wrote:

Stefanel, another question: if there will be a need to rebuild the tsvector columns, this should be able to be done without dropping and adding them again, right?

Actually, you're right. To force PostgreSQL to recalculate the tsvector__ values for every single row in the table, we can run a dummy update using the DEFAULT keyword:

```
UPDATE <table> SET tsvector__<field> = DEFAULT;
REINDEX TABLE CONCURRENTLY <table>;
ANALYZE <table>;
```

#6 - 10/08/2026 12:02 PM - Constantin Asofiei

Stefanel, please also look into if there is any performance impact by **not having** the tsvector columns as the last ones in the table.

#7 - 10/08/2026 01:02 PM - Ovidiu Maxiniuc

If new columns are added to a PSQL table, the fastest way to re-order the columns is using the following procedure:

- stop the FWD server;
- apply the transformations (adding column);
- for each affected table:
 - rename the table (it contain the current data and on which the alter table were performed):

```
ALTER TABLE abc RENAME TO abc_old;
```

- create the new table with the updated order of columns (from ddl/schema_table_<db>_postgresql.sql):

```
CREATE TABLE abc (recid INT8 NOT NULL, .... PRIMARY KEY (recid));
```

- move the data to new location:

```
INSERT INTO abc (recid, ...) SELECT recid, .... FROM abc_old;
```

- (re-)index the new table using the specific statements from ddl/schema_index_<db>_postgresql.sql;
- now, that the new table is fully functional, we can get rid of the old table:

```
DROP TABLE users_old;
```

- optionally, do a commit to limit the transaction size;
- if commit was not performed after processing each table, now it's the moment;
- restart FWD server.

This has the advantage that is 99% executed on server (SQL) side, with minimal communication with the client (no dump/restore is necessary), therefore is probably optimal from the PoV of the time FWD server is down.

The disadvantage is that each table needs to be processed individually, with DDLs extracted from target conversion, but these statements can be executed in parallel for multiple tables. Ultimately, a small script can be written to drive the process in full automatic fashion. It depends on the number of affected tables if that is feasible.

#8 - 10/09/2026 04:39 AM - Stefanel Pezamosca

- % Done changed from 0 to 10

- Status changed from New to WIP

I created branch 11956a from trunk/r16794 to work on this.
I'm currently doing some experiments to make a plan for the final implementation.

#9 - 10/09/2026 04:51 AM - Stefanel Pezamosca

Constantin Asofiei wrote:

Stefanel, please also look into if there is any performance impact by **not having** the tsvector columns as the last ones in the table.

Sequential scans will be slower by about 6-8% per tsvector column crossed when filtering on later columns, while index lookups, CONTAINS, and full-row reads show no difference

#10 - 10/09/2026 05:22 AM - Stefanel Pezamosca

Implementation plan

The tsvector__<field> columns will be placed right after the column(s) they index, in a new database and in a migrated one, so that an incremental ADD COLUMN leaves the schema identical to a fresh import. Rebuilding them will not move them. The minimum PostgreSQL version stays 12.

What I validated

I prototyped the migration in SQL and ran it on a copy of my testdb (27 tables, 11 with FTS columns, already migrated with the current trunk) and on a 1M-row table, built from the same words, on PostgreSQL.

Check	Result
Current layout	The 18 tsvector__* columns are always last. 7 of the 11 FTS tables already have the target layout, because their word-indexed field is their last field. Only 4 need a rewrite.
Rewrite of the 4 tables	Data, tsvector values and table definitions unchanged. Running it again does nothing.
Same as a fresh database	Column order, indexes and column definitions identical to a database created from the new DDL, for all 27 tables.
Database with word tables	Word tables, their triggers and trigger functions removed and the FTS columns added in one transaction, with the same values as today.
Later schema changes	A field added at the end, or a word index added later, gives the same layout as a fresh database.
Unsupported objects	A table with a view, an owned sequence or an FK from another table is refused and left unchanged.
Preserved objects	Extent-table FKs, grants and comments are kept.
Table in use	A table locked by another session fails after 5s and is left unchanged.
Check before COMMIT	Found two real gaps in the prototype (NOT NULL constraint names, comments) and rolled back both times.

Migration cost, 1M rows with 4 FTS columns:

Approach	Time	WAL
Today (4x ADD COLUMN, then the GIN indexes)	34.3s	1854 MB
Rewrite, then VACUUM ANALYZE	31.5s	888 MB

maintenance_work_mem = 1GB makes the rewrite 19% faster.

Rebuild cost, 1M rows, by share of stale rows:

Stale rows	Targeted UPDATE	Rewrite
1%	21.8s, 140 MB WAL	29.4s, 600 MB WAL
27%	36.2s, 1349 MB WAL, table +33%	37.6s, 666 MB WAL
100%	52.3s, 4.5 GB WAL, table 2x	29.8s, 1.1 GB WAL

A 1% sample estimated the stale share at 1.04% (exact: 1.00%) in 0.3s.

Not validated yet: the older PostgreSQL versions down to 12, and the Java code.

Correction to my note above: UPDATE <table> SET tsvector__<field> = DEFAULT does not recompute the column, the old value stays (checked on a real table). PostgreSQL recomputes a stored generated column only when one of its source columns is updated. The rebuild below relies on that.

1. Core DDL

- P2JPostgreSQLDialect: split getFtsColumnDDL() into a column definition (tsvector__f tsvector generated always as (...) stored).
- DDLGeneratorWorker.generateTableDDLImpl(): emit it inside CREATE TABLE right after the field's last column (the last __N element of an expanded extent) and remove the trailing alter table ... add column. GIN indexes and normalized extents are unchanged.

2. Migration (fts_migrate.sh / ImportWorker)

PostgreSQL cannot insert a column at a given position, so a table which is not in the target layout is rewritten, once, whatever its number of word indexes. This is the procedure Ovidiu described, with checks around it.

The FTS columns and their source columns come from the P2O, as ftsSourceColumns() does today. The target order is the live column order, with each FTS column right after its last source column. A table already in that order, with its GIN indexes, is skipped.

One transaction per table, in this order (the word-table triggers and FKs would otherwise make the table refused):

1. SET LOCAL lock_timeout = '5s', statement_timeout = 0 and maintenance_work_mem (sized by --max_threads), then lock the table.
 2. Drop the word tables with the existing getDropWordTableDDL().
 3. Refuse the table, unchanged, if it has anything the rewrite does not recreate: views or rules, triggers, policies, row level security, publications, extended statistics, owned sequences, identity columns, inheritance or partitions, a non-default tablespace or replica identity, FKs from tables other than <t>__N.
 4. Take a snapshot: column definitions, index and constraint definitions, FKs from <t>__N, owner, grants, comments, storage options, row count.
 5. Rename the table to <t>__fts_old and create <t> in the target order. Each column is written as format_type(), the collation name from pg_collation when it differs from the type's, not null and the default from pg_get_expr(). When the catalog has a named NOT NULL constraint for the column, the definition keeps that name (constraint <name> not null); the old table still holds the name at that point, so an unnamed definition would get a different one.
 6. INSERT ... SELECT the non-generated columns, drop the FKs from <t>__N and drop the old table (never CASCADE).
 7. Recreate the constraints (pg_get_constraintdef()), indexes (pg_get_indexdef()), GIN indexes, <t>__N FKs, owner, grants (aclexplode()) and comments (table, column, index, constraint).
 8. Compare with the snapshot; any difference other than the FTS columns means ROLLBACK.
- After the commit: VACUUM ANALYZE the table.
 - Before starting: refuse while other client sessions are connected (pg_stat_activity, backend_type = 'client backend'), unless --force is given. The largest tables go first.
 - --dry-run prints the tables, their target order and the refusals without changing anything (useful for Infor, who write their SQL upgrades by hand, #11842). --verify reports the tables which are not in the target layout.
 - One code path for every version from 12 on, without version checks: all statements and catalog objects used above exist in 12. The snapshot compares the columns through to_jsonb() of their pg_attribute rows, without attrelid and attnum, and without attahasmissing and attmissingval, which the rewrite clears for a column added later with a default. So the same query works everywhere and also covers the column settings newer versions add.

3. Rebuild (--rebuild)

Per table, the stale fraction (rows where tsvector__f IS DISTINCT FROM <expression>) is estimated from a 1% TABLESAMPLE SYSTEM; small tables are scanned fully.

- Nothing stale: skip.
- Up to about 5%: UPDATE <t> SET <source> = <source>, ... WHERE <any FTS column stale>, one statement for all FTS columns of the table. Only the stale rows are written and the table is not rewritten.
- Above that: the rewrite from section 2.

6. Tests

- Fresh DDL: plain, case-sensitive, expanded extent and last-field cases;

- Migration from word tables and from the current columns-at-the-end layout: the layout equals a fresh database (information_schema.columns across all tables); a re-run changes nothing.
- Refused, failed and locked tables leave the database unchanged; extent FKs, grants and comments survive.
- Rebuild after a UDF change, both paths: no stale values left, positions kept.

So, is the scan cost mentioned in [#11956-9](#) acceptable?

For a table the migration refuses, Should I skip it and finish with a non-zero exit code, or stopping the whole run?

Should anything in this plan be changed?

#11 - 10/09/2026 09:18 AM - Greg Shah

Before we move ahead, I want Alex and Eric to review and provide input.

Taking a performance hit on some sequential scans is certainly something we would not want, so it needs to be considered.

On the other hand, the maintenance problems introduced by using an at-the-end approach are real and will turn into production problems when not managed well in the future. Since customers implement their own schema editing, there are many changes for customers to accidentally edit the schema and leave it in an unexpected state that might not be found until much later.

#12 - 10/09/2026 09:19 AM - Greg Shah

Eric/Alex: Please provide your thoughts by EOD today.

#13 - 10/09/2026 10:22 AM - Stefanel Pezamosca

I also want to present both options fairly, why the tsvector columns are currently placed at the end of the table rather than next to their source columns. This is the usual practice, for the reasons outlined below.

1. **Storage and scan speed:** PostgreSQL aligns each value on a byte boundary which depends on its type. Fixed-width types (integer, timestamp) take less space and perform better when they are grouped together, so variable-length types like tsvector usually go at the end of the table, to keep the padding between columns low. A tsvector column in the middle also slows down the columns after it: to read a column, PostgreSQL has to step over every variable-length value stored before it in the row. I measured it on a 1M-row table: a sequential scan filtering on a column placed after tsvector columns is about 6-8% slower for each tsvector column it has to step over. Index lookups, CONTAINS and full-row reads showed no difference.
2. **DDL:** PostgreSQL has no ADD COLUMN ... AFTER <column>, unlike MySQL. Putting a column in the middle of an existing table needs a workaround, like rewriting the whole table or renaming columns. Adding it at the end is what ALTER TABLE ... ADD COLUMN does natively.
3. **Separation:** The tsvector columns are technical data, used only for the index.
Keeping them at the end leaves the application columns together and the schema easier to read for whoever inspects it.

Migration speed does not decide it. I measured both on a 1M-row table with 4 FTS columns: dropping and re-adding the tsvector columns in a single ALTER TABLE (Currently the code does one ALTER TABLE per column, not optimal) also rewrites the whole table, and it takes the same time as recreating the table with them next to their source, about 28s and 590 MB of WAL for both. The real difference is that the ALTER TABLE leaves the rest of the table as it is (constraints, indexes, foreign keys, grants, comments), while recreating the table means rebuilding all of them.

Eric/Alex: Please provide your thoughts by EOD today.

Personally, I think that tsvector is only one of multiple example of columns that are dynamic in size and are **not suitable** to be at the beginning of the table.

In general, fixed sized columns are put at the beginning so that offset computing is $O(1)$. If there are variable sized columns, then all columns after that have a dynamic algorithm to gather their offset. This is why it is preferred to have sized columns at the beginning.

I am not sure I understand the gain of sticking to the OE column order instead of hijacking that and using a FWD specific order of columns that maximizes performance. New columns are added to the end because it is cheap and fast, but on the log run, the table structure is the one granting query speed. Just like analyze, but for DDL, the table layout is the one that influences how fast query speed is.

Taking a performance hit on some sequential scans is certainly something we would not want, so it needs to be considered.

Well, it is a shame that **indexing** affects **table structure** and more precisely the seq scan. The table structure should be static and not influenced by what the user wants to index. If an index is added **later**, then it changes the table layout.

Can you reiterate **why** the `to_tsvector('simple', coalesce(series_title, ''))` is computed as a generated column in table and not **only in the word-index**? Other index lookups are technically also affected because:

- if you query on (non-word-index), you gather the rowid
- based on the rowid, you need to resolve the other columns in filters, sorting or select list
- if the columns in question (non-indexed) are after the computed column, they will be gathered **slower**.

So I don't this it is only about seq scan. An indexed scan (where you require fields after tsvector or use an unindexed column in the filter after the tsvector) may be slower.

This adds a persistent tsvector column that is automatically generated and updated by PostgreSQL whenever the row changes, so we don't need custom triggers. The index is built on this stored column.

I see:

```
CREATE INDEX textsearch_idx  
ON imdb_movies  
USING GIN (textsearchable_index_col);
```

but why not:

```
CREATE INDEX textsearch_idx  
ON imdb_movies  
USING GIN (to_tsvector('simple', coalesce(series_title, '')));
```

#16 - 10/12/2026 05:27 AM - Alexandru Lungu

PS: with the latest suggestion, the "computed" expression in question will occur **only in the textsearch_idx index** and won't affect the layout of the table, thus the performance of other indexes.

#17 - 10/12/2026 07:41 AM - Greg Shah

I am not sure I understand the gain of sticking to the OE column order instead of hijacking that

Consider that pretty much all customers are planning to leverage direct access to the converted RDBMS. Most already do this.

Also consider, many customers have code (or their own customers who have code) that accesses the existing database structure. This is most often done to enable reporting engines, user-level direct query capabilities and/or customizations/customer-specific extensions. As part of the conversion project, it is common for these external tools to need to be manually converted. The more hijacking we do, the bigger this job. It becomes a land mine waiting to explode, especially if we don't figure it out in advance.

Bottom line: just as everywhere else, we need to be as compatible as possible. **We must not** make such changes lightly.

#18 - 10/12/2026 07:41 AM - Greg Shah

Greg Shah wrote:

Eric/Alex: Please provide your thoughts by EOD today.

Eric?

#19 - 10/12/2026 07:48 AM - Alexandru Lungu

Also consider, many customers have code (or their own customers who have code) that accesses the existing database structure. This is most often done to enable reporting engines, user-level direct query capabilities and/or customizations/customer-specific extensions. As part of the conversion project, it is common for these external tools to need to be manually converted. The more hijacking we do, the bigger this job. It becomes a land mine waiting to explode, especially if we don't figure it out in advance.

That access using syntax like `select * from` that gathers the columns in the definition order I suppose.

Bottom line: just as everywhere else, we need to be as compatible as possible. We must not make such changes lightly.

Surely not. But I am trying to figure out if the internal representation of the table on disk (i.e. column order) is something to be compatible about or the migration from OE to PostgreSQL can help customers also reorganize the table structure for better performance.

Greg, I am not that confident on this idea **for this task**. However, I want to push on my suggestion on `USING GIN (to_tsvector('simple', coalesce(series_title, '')))`. I know we delivered a solution that stores that tsvector as a separate table column, but I want to understand the benefit of that. **Stefanel**, are we querying the `ts_vector` from Java? Isn't it used only for indexing purposes?

#20 - 10/12/2026 08:10 AM - Greg Shah

That access using syntax like `select * from` that gathers the columns in the definition order I suppose.

I would expect direct SQL access including dependencies on column positions.

#21 - 10/12/2026 08:22 AM - Stefanel Pezamosca

Alexandru Lungu wrote:

However, I want to push on my suggestion on USING GIN (to_tsvector('simple', coalesce(series_title, ''))); I know we delivered a solution that stores that tsvector as a separate table column, but I want to understand the benefit of that. **Stefanel**, are we querying the ts_vector from Java? Isn't it used only for indexing purposes?

I recall there were some downsides of using the expression only in the GIN index, but I'll double check. This could be a solution but I will do a comparison between both and come back with the results.

#22 - 10/12/2026 08:37 AM - Stefanel Pezamosca

Generated column vs. expression index for CONTAINS

- **A (current):** stored column tsvector__f = udf.words_to_tsvector(f, true), GIN index on the column.
- **B:** GIN index on udf.words_to_tsvector(f, true) directly, no column. PostgreSQL computes the value again whenever it needs it from the row.

PostgreSQL 18, default settings (work_mem = 4MB), real UDF, rows of 40 words.

Topic	A	B
Disk, 2M rows	1474 MB	732 MB
Columns after it (1M rows, scans over many rows)	7-9% slower	unchanged
Seen in select *	yes	no
Add to an existing table	table rewrite	CREATE INDEX only
Rebuild after a UDF change	UPDATE or rewrite	REINDEX
UPDATE of another column, 200k rows	4.0s	8.2s

Search, 2M rows (search text known to the planner / bind parameter):

Search	Rows matched	A	B
1 word	0.8%	19 / 8 ms	6 / 20 ms
Prefix	8%	341 / 434 ms	165 / 170 ms
OR of 20 words	15%	1440 ms	216 ms
OR of 100 words	55%	5.5 / 17.5 s	25.2 / 35.6 s
Prefix	~100%	0.43 / 1.3 s	43.7 / 44.1 s

- **Few matches:** B is as fast or faster, its table is 2.4x smaller. At 200k rows B was never slower.
- **Many matches on a big table:** B is 2-100x slower. Once the matching pages exceed work_mem, every row on them is checked again, and in B that calls the UDF for each row.
- **B needs the exact expression** in the query, otherwise the index is silently ignored (4.1s instead of 0.5ms).

Conclusion: B is smaller, leaves the table unchanged and is as fast or faster when a search matches up to ~15% of the rows. A stays fast when a search matches most of a large table, where B took up to 44s. From what I understand, work_mem directly sets the point where B slows down a lot: B stays fast while the heap pages holding the matches fit in work_mem.

The expression index for CONTAINS may be worth using, so I can test it further.

#23 - 10/12/2026 08:49 AM - Alexandru Lungu

Conclusion: B is smaller, leaves the table unchanged and is as fast or faster when a search matches up to ~15% of the rows. A stays fast when a search matches most of a large table, where B took up to 44s. From what I understand, `work_mem` directly sets the point where B slows down a lot: B stays fast while the heap pages holding the matches fit in `work_mem`.

This is a general "breakpoint" for any kind of query. If the analysis detects that the number of rows going to match exceeds a certain threshold, it will fallback to the seq scan (or other index) scan. In this case, the expression covered in the index is no longer "cached" in the index, but recomputed. Basically, it becomes a "slow filter".

Prefix ~100% 0.43 / 1.3 s 43.7 / 44.1 s

This is odd.

Eric/Constantin please provide additional feedback.

#24 - 10/12/2026 08:57 AM - Constantin Asofiei

Alexandru Lungu wrote:

Eric/Constantin please provide additional feedback.

Considering not having the tsvector column is a performance downgrade, is a no-go for me.

So, I think we need to establish the process, which must include:

- migrating an existing production database from word indexes to FTS (following the idea Ovidiu mentioned in [#11956-7](#))
- adding a new tsvector column to a production database
- dropping a tsvector column from a production database

Also, from what Stefanel posted, the same problem may also apply to other computed column indexes, like for case-insensitive fields. Stefanel, please do also a performance test related to this:

- index with trunk, where the index has the computed column
- the computed column exists in the table, and the index is on that column

#25 - 10/12/2026 09:01 AM - Alexandru Lungu

Considering not having the tsvector column is a performance downgrade, is a no-go for me.

Well, it is a performance downgrade for some cases and improvement for others:

Search	Rows matched	A	B
OR of 20 words	15%	1440 ms	216 ms
OR of 100 words	55%	5.5 / 17.5 s	25.2 / 35.6 s

#26 - 10/12/2026 09:03 AM - Alexandru Lungu

Also:

Columns after it (1M rows, scans over many rows) 7-9% slower unchanged

Is favorable improvement for B.

#27 - 10/12/2026 09:08 AM - Constantin Asofiei

Alex/Stefanel: is 'B' (index with computed GIN index) dependent on 'work_mem' set at postgresql? If so, we need to consider that this is per JDBC connection, right (here I mean FWD client)?

And how can this scale when we have 100s of clients? How much additional RAM is required for the database server, if we increase this to a value where it can handle not 2M rows, but 200M or more rows?

Although I don't think a client can request in 'normal operation conditions' a SELECT to retrieve more than lets say in the order of 10k rows.

#28 - 10/12/2026 09:10 AM - Constantin Asofiei

PS: when talking about performance, I'm inclined to consider almost always the 'worst case scenario'; although the discussion after this is how probable the 'worst case scenario' to happen in real production use-cases.

#29 - 10/12/2026 09:21 AM - Alexandru Lungu

Constantin Asofiei wrote:

PS: when talking about performance, I'm inclined to consider almost always the 'worst case scenario'; although the discussion after this is how probable the 'worst case scenario' to happen in real production use-cases.

The thing with [#11956-22](#) comparison shows that B is worse because the UDF is evaluated more times, where A is cached. So, we are in a scenario where we have to pick:

- the caching overhead for small cases is making things slower overall.
- but the caching is sometimes better as we avoid some time recomputing the words of a phrase.

But this time, the caching is in the DB **and persistent**.

Stefanel, I wonder if we can make words_to_tsvector faster so that we nullify completely the need of caching it in a column. There are two steps:

- `regexp_split_to_array('some text', '([?[:-.])(?!\d)(?![-.])({delimiters}))+' );` - this is implemented in PG core in C, so I think for decently sized texts, it should be fast.

```
coalesce(array_to_tsvector(ARRAY(  
    SELECT word  
    FROM unnest(udf.split_words('some text', true)) AS word  
    WHERE octet_length(word) BETWEEN 1 AND 2046  
)), ''::tsvector)
```

- this second udf may be performance killer because it has to transform the array in a table, back in an array and then in a tsvector. This extra work is just to trim the words that are bigger than 2046 bytes. From my POV, this is outside the "decently sized text". These texts **won't exist in the column anyway**, because they will be toasted.

So, maybe we can distinguish between long texts and short texts:

- if `octet_length(txt) <= 2046`, then the where is irrelevant.
- if `octet_length(txt) > 2046`, then we can risk having very big words. We need to think more on this.

Is OE storing any arbitrary sized texts in the word indexes or is throwing an error? Is this filtering mimicking a OE behavior that drops words that are bigger than 2046?

#30 - 10/12/2026 09:25 AM - Stefanel Pezamosca

The "2046" is a postgresql tsvector limit for one lexeme(word). If ever happens the insert/update will just fail with an error.

#31 - 10/12/2026 09:33 AM - Alexandru Lungu

Stefanel Pezamosca wrote:

The "2046" is a postgresql tsvector limit for one lexeme(word). If ever happens the insert/update will just fail with an error.

Oh god. I understand. Yet, this is something we can preprocess in FWD before insert/update I suppose, so we don't overwhelm the DB to do that. With this, we can emit an error condition specific to FWD so that we don't silently corrupt data, but actually leverage the business logic converted to error handle this.