FUJITSU

shaping tomorrow with you

Pluggable table access methods

Haribabu Kommi / Pankaj Kapoor

About us

FUJITSU

Haribabu Kommi

- Working for Fujitsu Australia Software Technology
- 11+ years of experience in development of database
  - In memory, &
  - Disk based
- Working in PostgreSQL ecosystem from around 7+ years
  - Contributions include reviews, bug-fixes, feature development of core server
  - and in various tools/drivers e.g. JDBC

Pankaj Kapoor

- Working for Fujitsu Australia Software Technology
- 15+ years of experience in diverse domains ranging from applications to telecom
- Leading deliveries of Fujitsu Enterprise Postgres (FEP) from Australia.
- Working in PostgreSQL ecosystem from around ~3 years

Contents

FUJITSU

- History of the development
- People behind it
- High level overview
  - what is pluggable table access method
  - why not FDW
  - what is supported in v12
- New syntax support
- Overview of table access method API
- Possibilities using pluggable table AM
- Further development

History of the development

FUJITSU

- The seed was sown by **Alvaro**
  - Columnar Storage was requested and fairly invasive patch was worked upon around Aug-2016
  - It was agreed that whole database can not be columnar based as it will serve OLAP but will hit OLTP scenarios
- Fujitsu also wanted to contribute their columnar storage - Vertical Clustered Index (VCI). To support the cause, **Haribabu,** from Fujitsu Australia Software Technology participated in the development.
- **Robert Haas** suggested to work on rather generic and extensible approach - pluggable table access methods; rather than just focussing on storage layer
- With huge efforts from **Andres** in terms of development, reviews, fixes and guidance, Pluggable Table AM got committed in Mar 2019

People behind it

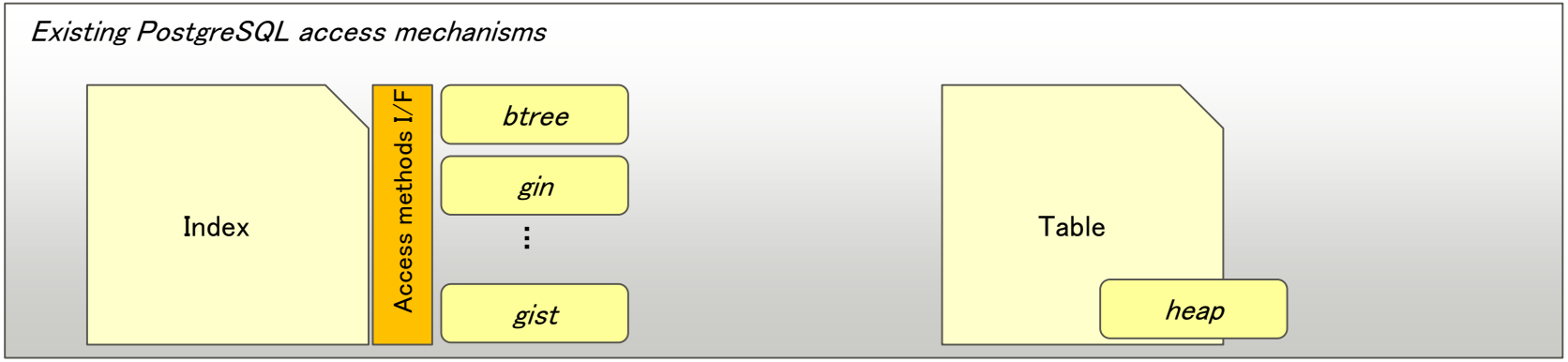
FUJITSU

- Pluggable Table AM is a big feature and many people were involved in completion of this feature. The list of people involved were:
  - Andres Freund
  - Haribabu Kommi
  - Alvaro Herrera
  - Alexander Korotkov
  - Dimitri Dolgov
  - Ashutosh Bapat
  - Amit Khandekar
  - David Rowley
  - and others
- Draft Pluggable Table AM was being used by zHeap team, thereby helping to identify needs apart from what Heap AM needs

What is pluggable table access method

FUJITSU

- Till PostgreSQL 11, core maintains a pluggable access mechanism for Index to choose different index implementation; however no similar mechanism was available for Table



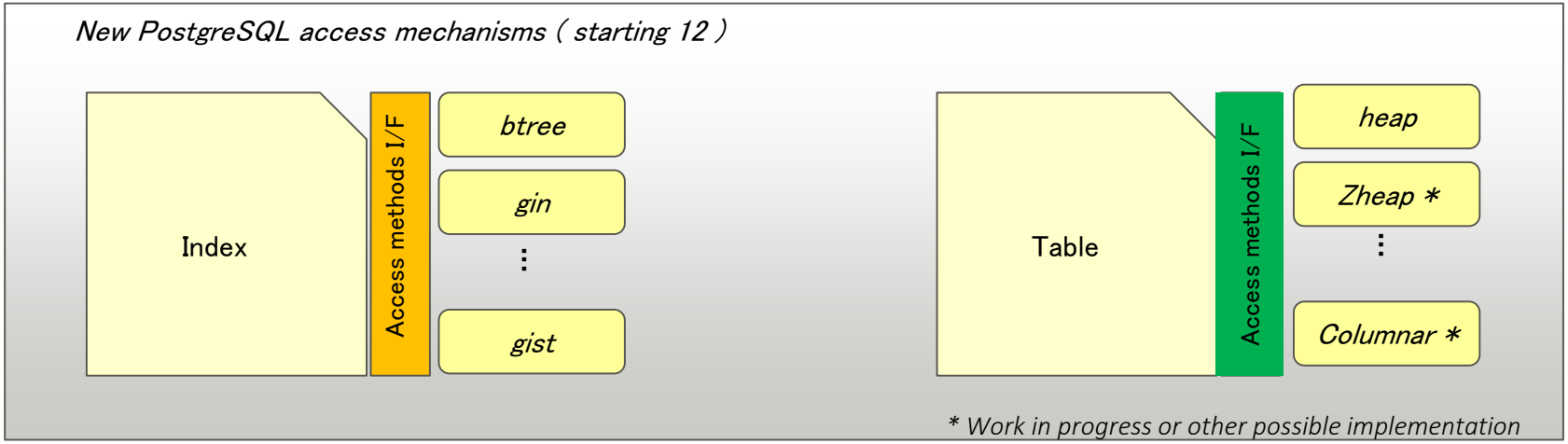
- The idea was to implement storage access mechanism (similar to index) for Tables as well to allow different tuple storage mechanisms. The same is been referred to Pluggable table access methods henceforth.

- Pluggable table access methods exposes API, that facilitates any intendent developer/team to generate specialised storage/access mechanisms of tuples for a table.

What is pluggable table access method?

FUJITSU

- Pluggable table access methods API implementation also includes moving original heap mechanism over pluggable access methods, which will be the default available access method.



- There are many organizations that are working on implementing their own tuple storage types on top of pluggable access methods API, hopefully PG v13 will have some ready for integration.

Why not FDW

FUJITSU

- FDW is intended to access foreign data and not for storing local data
  - using this approach to access other storage types cannot yield best results and will have its own limitation e.g. DDL support

- Couple of are columnar storage extensions that are available for PostgreSQL using FDW
  - cstore_fdw
  - clickhousedb_fdw - ‘might’ be the right usage/hack, as it pushes/access data on ClickhouseDB server. But question remains, do we need a different **server** to store columnar data ?

What is supported in v12

FUJITSU

- User can create new TABLE type access methods
- TABLE access methods can be assigned to
  - Tables, and
  - Materialised Views
- Indexes and its access methods remains same. No changes have been done for the same
- The supported API is currently useful to create row based table storages
  - This however can be contended; and
  - Doesn’t stop developers to hack it for columnar storage as well

New syntax support | User's view

FUJITSU

- CREATE ACCESS METHOD *<new access method>* TYPE TABLE HANDLER *<table_am_handler>*
- CREATE TABLE ... USING *<new access method>* ...
- CREATE TABLE ... USING *<new access method>* AS ...
- CREATE MATERIALIZED VIEW ... USING *<new access method>* ...

Overview of Table AM API | Developer's view

FUJITSU

- The following structure contains all the APIs that are necessary for any pluggable table access method provider. The handler function must return the filled API structure.

```
/*
 * API struct for a table AM.  Note this must be allocated in a
 * server-lifetime manner, typically as a static const struct, which then gets
 * returned by FormData_pg_am.amhandler.
 *
 * In most cases it's not appropriate to call the callbacks directly, use the
 * table_* wrapper functions instead.
 *
 * GetTableAmRoutine() asserts that required callbacks are filled in, remember
 * to update when adding a callback.
 */
typedef struct TableAmRoutine
{
...
}
```

Overview (cont.)

FUJITSU

- A total of 38 API's are available in the TableAmRoutine structure, except Bitmap and bulk insert API's, rest are all mandatory to develop a new table access method.

- There are 10 different categories of API that are present in the TableAmRoutine that needs to supplied by the new access methods.
  - Slot related callbacks –
    - Callback is to provide the slot type that is used by the AM
  - Table scan callbacks –
    - Callbacks to perform scanning of a relation and provide the necessary tuples
  - Parallel table scan callbacks –
    - Callbacks to perform scanning of a relation using parallel workers to speed up the relation scan.
  - Index scan callbacks –
    - Callbacks to perform scanning of a relation from the index

Overview (cont.)

FUJITSU

- Non-modifying tuple callbacks –
  - Callbacks to check the tuple, like tuple visibility and etc.
- Modifying tuple callbacks –
  - Callbacks to modify the tuple, like insert, update and etc.
- DDL callbacks –
  - Callbacks that handle the operations like setting the relfilenode, vacuum and etc.
- Misc callbacks –
  - Callbacks to provide AM specific information like toast and etc.
- Planner callbacks –
  - Callback to provide relation estimation size.
- Executor callbacks –
  - Callback for bitmap and sample scan functionality.

Possible implementation over pluggable table AM

FUJITSU

- Using the pluggable table access methods, it is possible to implement many different variety of table AM's, such as:
  - An alternative to heap (zHeap)
  - Columnar table
  - In-memory table
  - Index organized table
  - Etc.

An alternative to heap - zHeap

FUJITSU

- zHeap is already in progress with following objectives
  - Provide better control over bloat using in-place updates and undo records for delete
  - Reduce write amplification as compared to heap
  - Reduce tuple size by reducing tuple header
- We expect zHeap to be available in community version by PG v13
- Currently supported pluggable table access methods meets basic needs of zHeap (not all though)

Columnar table

FUJITSU

- The storage layout of the columnar table is column-wise instead of row-wise.

- Fujitsu also has its own time-tested columnar table that support both write and read query performance thereby achieving OLAP and OLTP cases. In past it needed core level changes – however now getting considered over Pluggable Table AM

- Zedstore, which is under development based on pluggable table access methods supports columnar table using btree index.

- With pluggable Table AM, the implementations will be rather focussed without bothering to change the core server

In-memory table

FUJITSU

- Main memory is used as primary storage tables. Rows in the table are read from and written to memory. Some considerations:
  - Copy of the table data is maintained on disk, only for durability
  - Follow Only in memory table (MongoDB has in-memory storage engine)
- in_memory is an extension in an enterprise version of postgres implemented via FDW. Implementing it over Pluggable API will reduce its codebase, let reap benefits of core and more
- Existing heap like mechanism, sans buffer-manager, will give an in-memory table AM; but it may need some additional capability to achieve syncing for persistence (if required).

Index organized table

FUJITSU

- In **index-organized table** unlike ordinary (heap-organized) table whose data is stored as an unordered collection (heap), data for an index-organized table is stored in a B-tree index structure in a primary key sorted manner. Each leaf block in the index structure stores both the key and non-key columns.

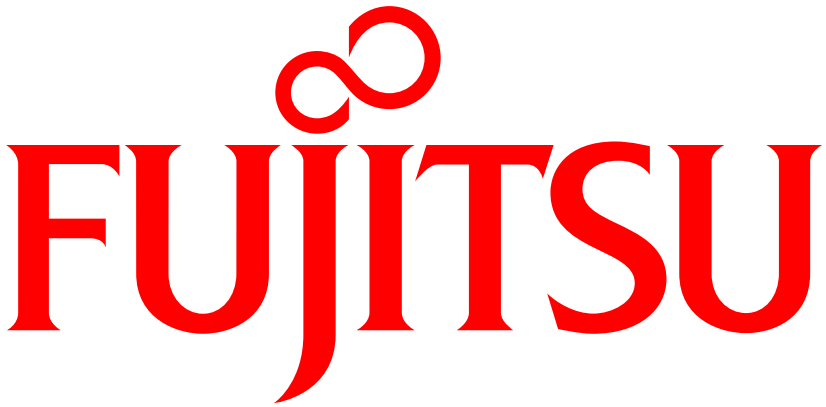
- This is some what similar to INCLUDE column support in PostgreSQL, but it eliminates the needs two place storage.

- Some quick implementation is possible by reusing zedstore columnar implementation, by storing all columns as part of the index organized by a primary key column.

Further development

FUJITSU

- New API to share targetlist columns from the table during the select operation, so that the specified columns can only be returned
- ALTER table syntax enhancement to switch from one AM to another
- Adding cost functions to let the planner know more about Table AM
- Not exactly related to pluggable table AM, but following will open up more avenues
  - Executor batching
  - Executor vectorization



shaping tomorrow with you