Go Faster With Native Compilation

PGCon 2015

18th June 2015

KUMAR RAJEEV RASTOGI (rajeevrastogi@huawei.com)
PRASANNA VENKATESH (prasanna.venkatesh@huawei.com)
TAO YE (yetao1@huawei.com)

Who Am I?

- KUMAR RAJEEV RASTOGI
  - Senior Technical Leader at Huawei Technology for almost 7 years
  - Have worked to develop various features on PostgreSQL (for internal projects) as well as on other In-House DB.
  - Active PostgreSQL community members, have contributed many patches.
  - Holds around 12 patents in my name in various DB technologies.
  - I have presented two papers in India PGDay (2014 and 2015).
  - Prior to this, worked at Aricent Technology for 3 years.

Blog - rajeevrastogi.blogspot.in

LinkedIn - http://in.linkedin.com/in/kumarrajeevrastogi

Agenda

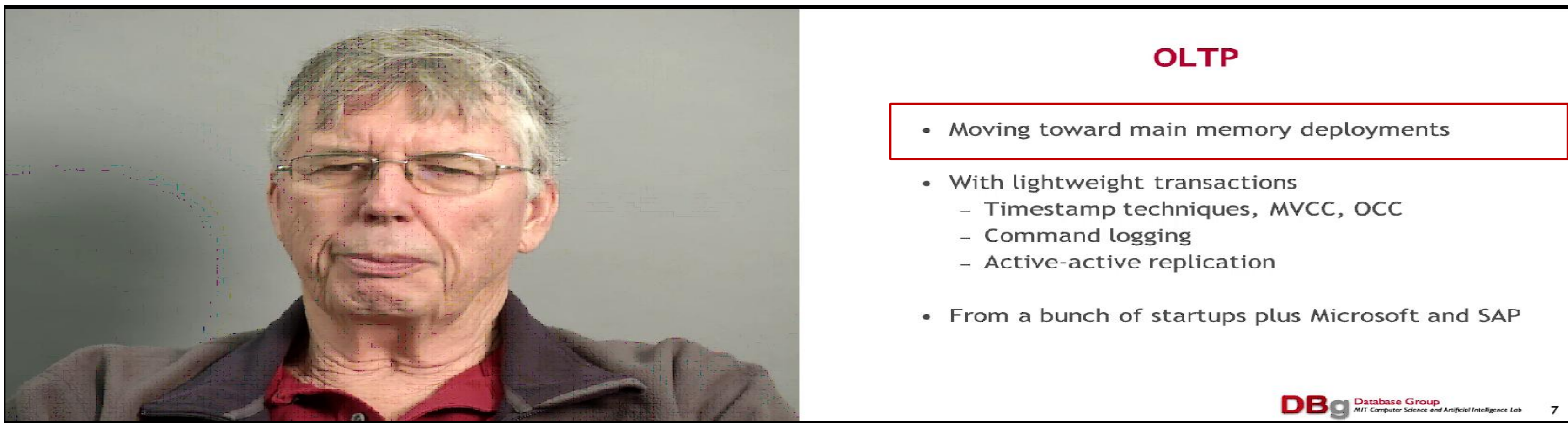
1. Background
2. Current Business Trend
3. Native Compilation
4. Cost model
5. What to Compile
6. Schema binding
7. Schema binding Solution
8. Performance Scenario
9. Conclusion

Background

The traditional database executors are based on the fact that "I/O cost dominates execution". These executor models are inefficient in terms of CPU instructions.

Now most of the workloads fits into main memory, which is consequence of two broad trends :

1. Growth in the amount of memory (RAM) per node/machine
2. Prevalence of high speed SSD



Source: ICDE Conference

So now biggest bottleneck is CPU usage efficiency not I/O. Our problem statement is to make our database more efficient in terms of CPU instructions – there by leveraging the larger memory

Current Business Trend

Slowly database industries are reaching to a point where increase of throughput has become very limited. Quoting from a paper on Hekaton - *The only real hope to increase throughput is to reduce the number of instructions executed but the reduction needs to be dramatic. To go 10X faster, the engine must execute 90% fewer instructions and yet still get the work done. To go 100X faster, it must execute 99% fewer instructions.*

Such a drastic reduction in instruction without disturbing whole functionality is only possible by code specialization (a.k.a Native Compilation or famously as LLVM) i.e. to generate code specific to object/query.

Current Business Trend Contd…

Many DBs are moving into compilation technology to improve performance by reducing the CPU instruction some of them are:

- Hekaton (SQL Server 2014)
- Oracle
- MemSQL

Transac-tion size in #lookups	CPU cycles (in millions)		Speedup
	Interpreted	Compiled	
1	0.734	0.040	10.8X
10	0.937	0.051	18.4X
100	2.72	0.150	18.1X
1,000	20.1	1.063	18.9X
10,000	201	9.85	20.4X

Hekaton: Comparison of CPU efficiency for lookups

Source: Hekaton Paper

Native Compilation

Native Compilation is a methodology to reduce CPU instructions by executing only instruction specific to given query/objects unlike interpreted execution. Steps are:

1. Generate C-code specific to objects/query.
2. Compile C-code to generate DLL and load with server executable.
3. Call specialized function instead of generalized function.

Source: ICDE Conference

e.g. Expression: **Col1 + 100**

Traditional executor will requires **100's of instruction** to find all combination of expression before final execution, whereas in vanilla c code, it can directly execute in **2-3 instructions**.

Cost model

Cost model of specialized code can be expressed as:

cost of execution = generate specialized code
+ compilation
+ execute compiled code

Execution of compiled code is very efficient but generation of specialized code and compiling same may be bit expensive affair. So in order to drive down this cost:

1. Generate and compile the code once and use it many times; this distributes the constant cost.
2. Improve the performance of generation and compilation significantly.

What to Native Compile?

Any CPU intensive entity of database can be natively compiled, if they have similar pattern on different execution. Some of the most popular one are:

- Schema (Relation)
- Procedure
- Query
- Algebraic expression

Note: We will target only Schema for this presentation.

Schema binding

Property of each relation:

1. Number of attributes, their length and data-type are fixed.
2. Irrespective of any data, it is going to be stored in similar pattern.
3. Each attributes are accessed in similar pattern.

Disadvantage of current approach for each tuple access:

1. Loops for each attribute.
2. Property of all attributes are checked to take many decisions.
3. Executes many unwanted instructions.

Schema binding Contd…

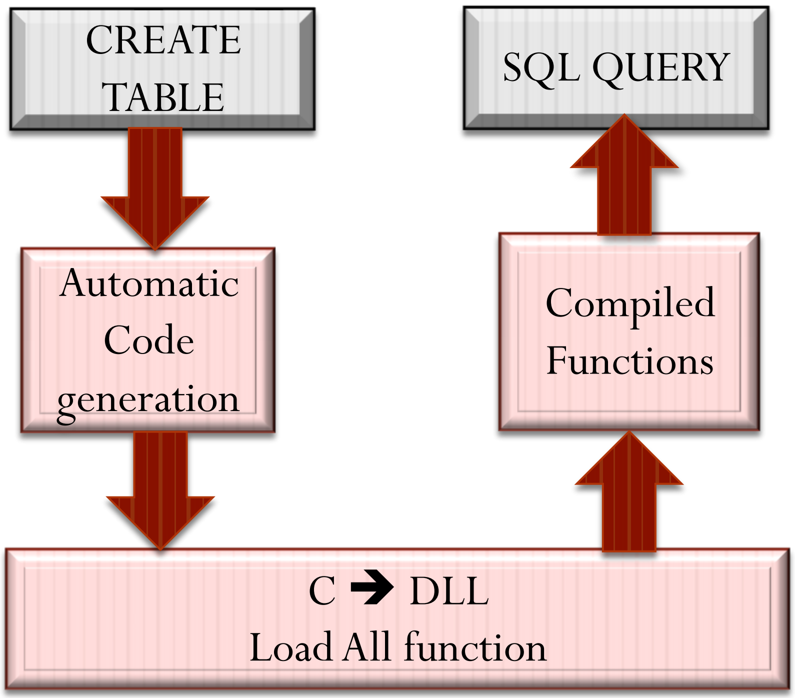
So we can overcome the disadvantage by natively compiling the relation based on its property to generate specialized code for each functions of schema.

Schema Binding = Native Compilation of Relation

Benefit:

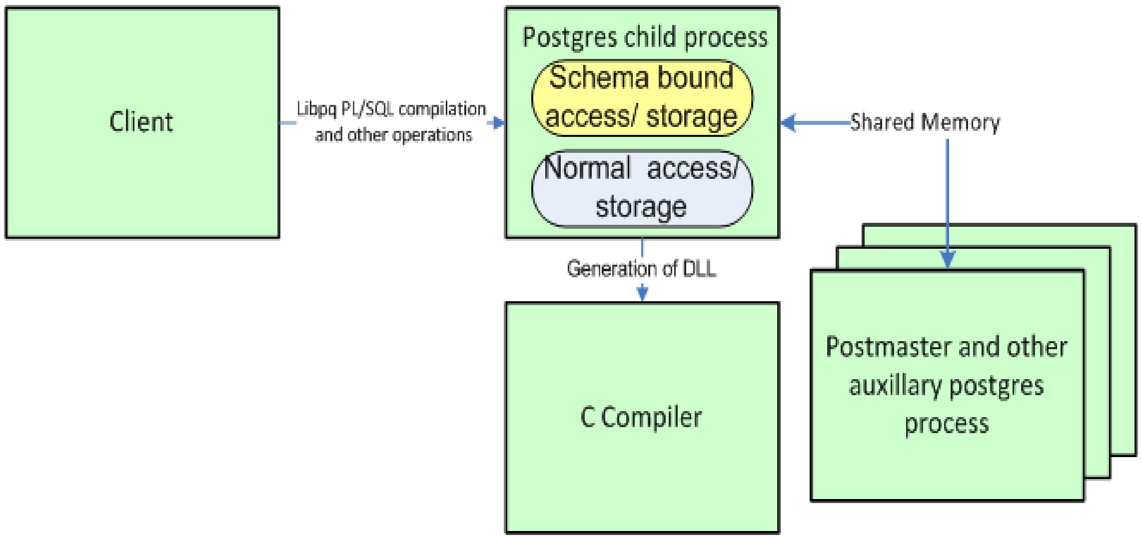
1. Each attribute access gets flattened.
2. All attribute property decision are taken during code generation.
3. No decision making at run-time.
4. Reduced CPU instruction.

Schema binding Contd…



Once a create table command is issued, a C-file with all specialized access function is generated, which is in turns gets loaded as DLL. These loaded functions are used by all SQL query accessing the compiled table

Schema binding Contd…

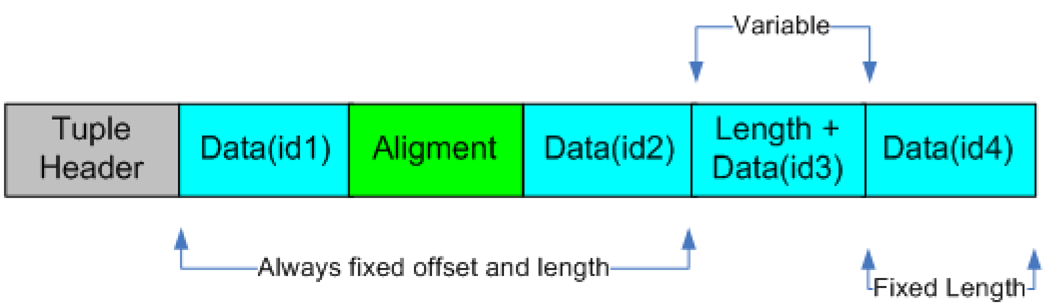


This show overall interaction with schema bound. Any query issued from client can use schema bound function or normal function depending on the underlying table.

Schema binding: Example

Schema:

create table tbl (id1 int, id2 float,
id3 varchar(10), id4 bool);



Field id1 and id2 is going to be always stored at same offset and with same alignment, no change at run time. Only variable length attribute and attribute following this will have variable offset.

Schema binding: Example

Using current approach:

```
if (thisatt->attlen != -1)
{
    offset = att_align_nominal(off, thisatt->attalign)
    values[1] = fetchatt(thisatt, tp + offset)
    offset = att_addlength_pointer(off, thisatt->attlen,
                                   tp + off);
}
```

Each Line here is macro, which invokes multiple condition check to decide the action

Access Using specialized code:

method-1:

```
values[1]  = ((struct tbl_xxx*)tp)->id2;
```

method-2:

```
offset = DOUBLEALIGN(offset);
values[1] = *((Datum *)(tp + offset));
offset += 8;
```

See details about this in further slides.

Conclusion: *Specialized code uses fewer number of instruction compare to generalized code and hence better performance*.

Schema Binding Solution

Solution can be categorized as:

1 → Opting for schema bind.

2 → Functions to be customized.

3 → Customized function generation.

4 → Loading of customized function.

5 → Invocation of customized function.

6 → How to generate dynamic library.

Solution: Opting for schema bind tuple

CREATE [[GLOBAL | LOCAL] { TEMPORARY | TEMP } | UNLOGGED] TABLE [IF NOT EXISTS] table …[TABLESPACE tablespace_name] **[SCHEMA_BOUNDED]**

SCHEMA_BOUND is new option with CREATE TABLE to opt for code specialization.

Solution: Functions to be customized

Function Name (xxx → relname_relid)	Purpose
heap_compute_data_size_xxx	To calculate size of the data part of the tuple
Heap_fill_tuple_xxx	To fill the tuple with the data
Heap_deform_tuple_xxx	Deform the heap tuple
Slot_deform_tuple_xxx	To deform the tuple at the end of scan to project attribute
Nocachegetattr_xxx	To get one attribute value from the tuple for vacuum case

Solution: Function Generation

Customized function for tuple access of a table can be categorized in 3 approaches:

Method-1 → With Tuple format change

Method-2 → Without changing the tuple format.

Method-3 → Re-organize table columns internally to make all fixed length and variable length attribute in sequence.

Solution: Function Generation-Method-1

A structure corresponding to relation will be created in such a way that each attribute's value/offset can be directly referenced by typecasting the data buffer with structure.

e.g. Consider our earlier example table:

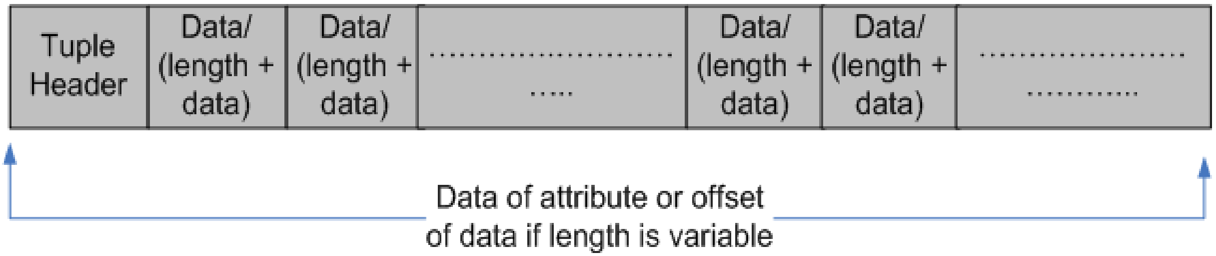
```
create table tbl (id1 int, id2 float, id3 varchar(10), id4 bool);
```

```
typedef struct schemaBindTbl_xxx
{
        int id1;
        float id2;
        short id3_offset;
        bool id4;
        /* Actual data for variable size column*/
} SchemaBindTbl_xxxx;
```

Structure member variable id1, id2 and id4 contains actual value of column, whereas id3_offset stores the offset of the column id3, as during create table it is not known the size of the actual value going to be stored. End of this structure buffer will hold data for variable size column and it can be accessed based on the corresponding offset stored.

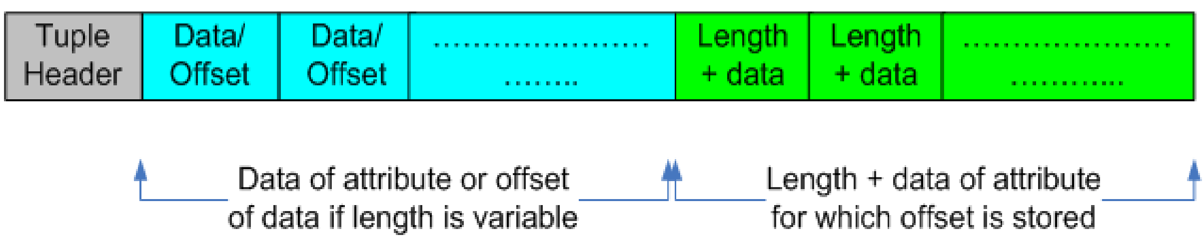
Solution: Function Generation-Method-1 Contd…

Existing Tuple Format



All attribute values stored in sequence.

New Tuple Format



Value of fixed length attribute but offset of variable length attribute stored in sequence. So structure typecast will give either value or offset of value.

Solution: Function Generation-Method-1 Contd…

So using this structure, tuple data can be stored as:

Fixed size data-type storage:

```
((SchemaBindTbl_xxxx*)data)->id1 = DatumGetXXX(values[attno]);
```

Variable size data-type storage:

```
((SchemaBindTbl_xxxx*)data)->id3_offset = data_offset;
data_length = SIZE((char*)values[attno]);
SET_VARSIZE_SHORT(data + data_offset, data_length);
memcpy(data + data_offset + 1, VARDATA((char*)values[attno]), data_length -1);
data_offset += data_length;
```

Using this approach heap_fill_tuple function can be generated during create table.

Solution: Function Generation-Method-1 Contd…

Similarly, each attribute value from tuple can be accessed as:

Fixed size data-type access:

```
values[attno]  = ((SchemaBindTbl_xxxx*)data)->id1;
```

Variable size data-type access:

```
data_offset = ((SchemaBindTbl_xxxx*)data)->id3_offset ;
values[attno] = PointerGetDatum((char *) ((char*)tp + data_offset));
```

Using this approach all function related to deformation of tuple (i.e. heap_deform_tuple, slot_deform_tuple and nocachegettr) can be generated during create table.

Solution: Function Generation-Method-1 Contd…

Advantage:

1. No dependency on previous attributes.
2. Any of the attribute value can be accessed directly.
3. Access of attribute value is very efficient as it will take very few instructions.

Disadvantage:

1. Size of the tuple will increase leading to more memory consumption.

Solution: Function Generation-Method-2

This method generates the customized functions without changing the format of the tuple.

This approach uses slight variation of existing macros:

- fetch_att
- att_addlength_pointer
- att_align_nominal
- att_align_pointer

These macros takes many decision based on the data-type, its size of each attributes which is going to be same for a relation.

So instead of using these macro for each tuple of a relation at run-time, it is used once during table schema definition itself to generate all customized function.

Solution: Function Generation-Method-2 Contd…

So as per this mechanism, code for accessing float attribute will be as below:

```
offset = DOUBLEALIGN(offset);               → Skipped alignment check
values[1] = *((Datum *)(tp + offset));      → Skipped datum size check
offset += 8;                                → Skipped attribute length check
```

Similarly access for all other data-type attributes can also be generated.

Using the combination of other macro, customized code for all other functions used for tuple access can be generated.

Solution: Function Generation-Method-2 Contd…

Advantage:

1. Existing tested macro are used, so it is very safe.
2. No change in tuple format and size.
3. Reduces number of overall instruction by huge margin.

Disadvantage:

1. Dependency on previous attribute incase previous attribute is variable length.

Solution: Function Generation-Method-3

This method is intended to use advantages of previous methods i.e.

- ***Make least number of attribute dependency***

  All fixed length attributes are grouped together to make initial list of columns followed by all variable length columns. So all fixed length attributes can be accessed directly. Change in column order will be done during creation of table itself.

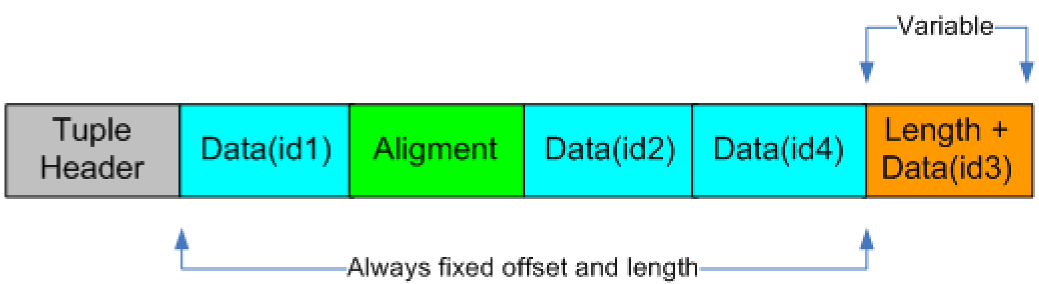
- ***No change in tuple size, so access of tuple will be very efficient***

  In order to achieve this, we use Method-2 to generate specialized code.

Solution: Function Generation-Method-3 Contd…

E.g. Consider our earlier example:

create table tbl (id1 int, id2 float, id3 varchar(10), id4 bool);



So in this case, while creating the table id1, id2 and id4 will be first 3 columns followed by id3.

So access code can be generated directly during schema definition without dependency on any run time parameter because all of the attribute offset is fixed except of variable length attributes.

If there are more variable length attributes then they will be stored after id3 and for them it will have to know the length of the previous columns to find the exact offset.

Solution: Function Generation-Method-3 Contd…

Advantage:

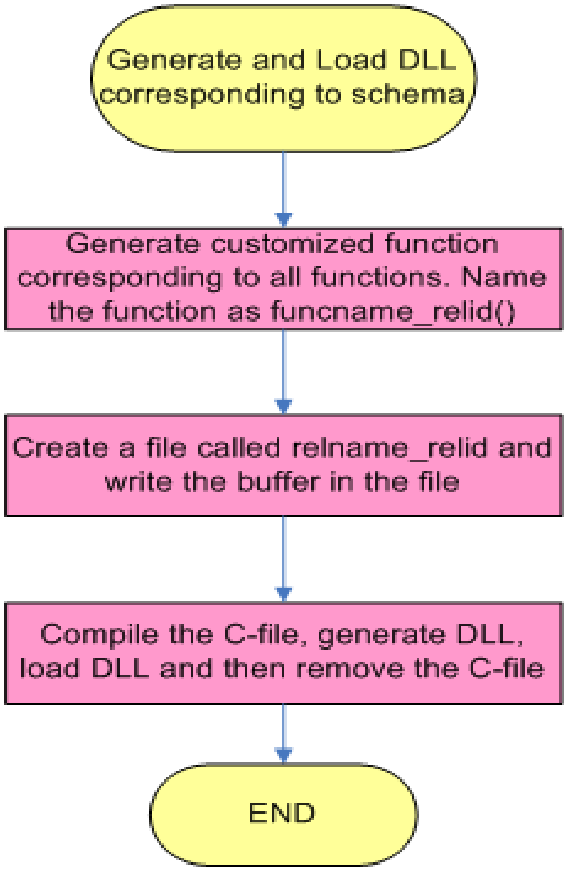
1. Existing tested macro are used, so it is very safe.
2. No change in tuple format and size.
3. Reduces number of overall instruction by huge margin.

Disadvantage:

1. There will be still dependency among multiple variable length attributes (if any).

Solution: Loading of customized functions

Once we generate the code corresponding to each access function, the same gets written into a C-file, which in turn gets compiled to dynamic linked library and then this dynamic library gets loaded with server executable. So now any function of the library can be invoked directly from the server executables.



Solution: How to generate dynamic library

The generated C-file should be compiled to generate dynamic library, which can be done using:

1. *LLVM*

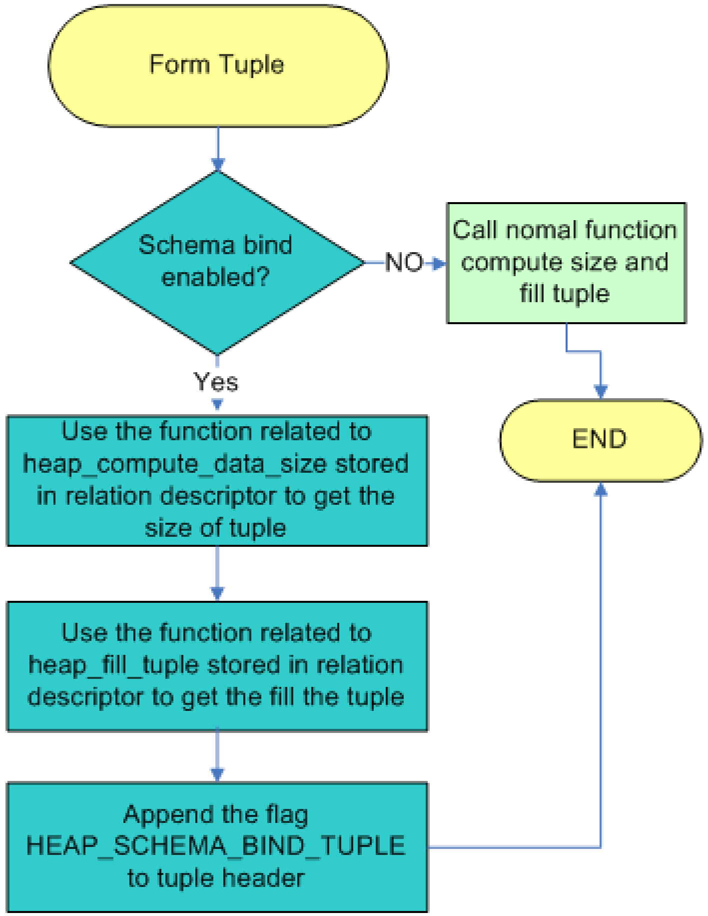
   Compilation using the LLVM will be very fast.

2. *GCC*

   GCC is standard way of compiling C file but it will be slow compare to LLVM.

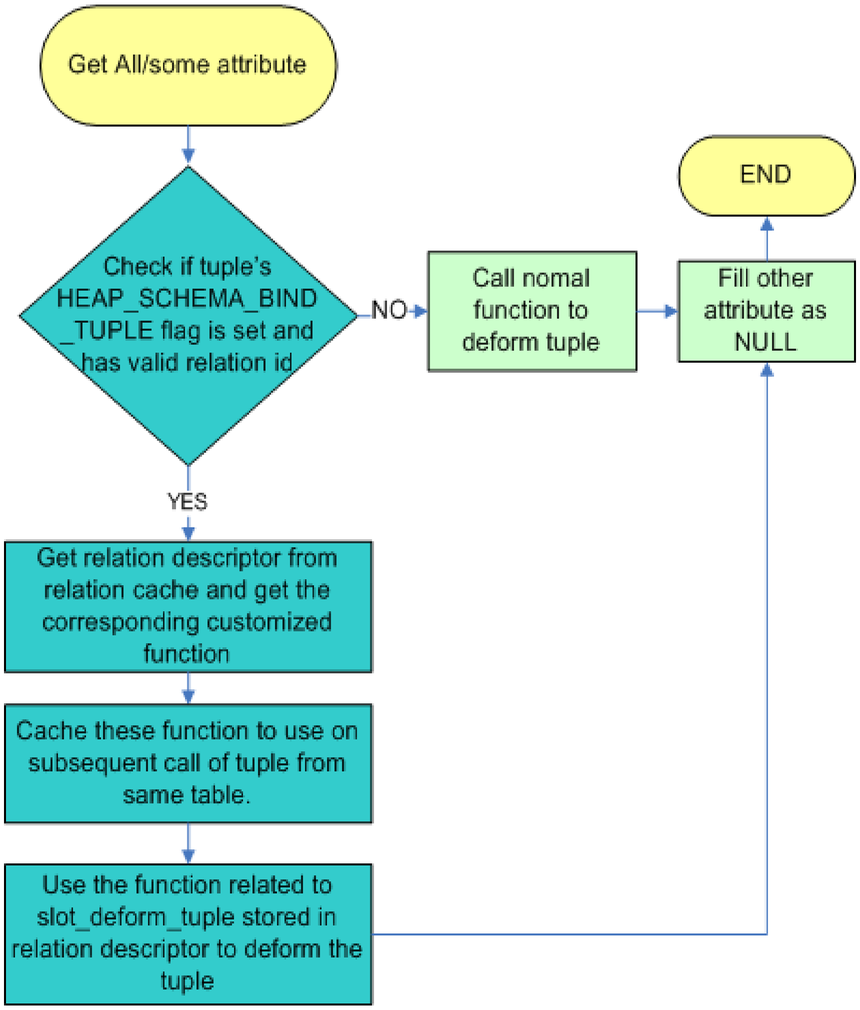
Solution: Invocation of Storage Customized Function

While forming the tuple, corresponding relation option schema_bound will be checked to decide whether to call customized function corresponding to this relation or the standard generalized function. Also in tuple flag t_infomask2, HEAP_SCHEMA_BIND_TUPLE (with value 0x1800) will be appended to mark the schema bounded tuple.



Solution: Invocation of access customized function

The tuple header's t_infomask2 flag will be checked to see , if HEAP_SCHEMA_BIND_TUPLE is set to decide whether to call customized function corresponding to this relation or the standard generalized function.



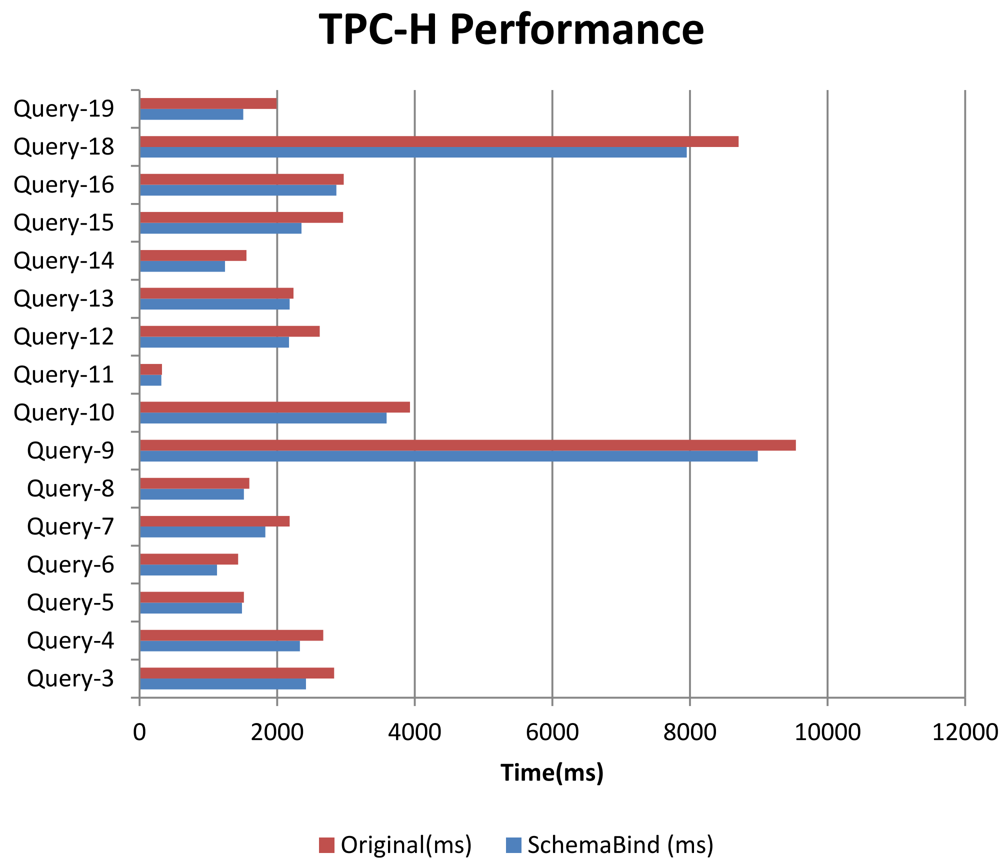
Performance (TPC-H):

The system configuration is as below:

SUSE Linux Enterprise Server 11 (x86_64), 2 Cores, 10 sockets per core

TPC-H Configuration: **Default**

TPC-H Query	Improvement(%)
Query-1	2%
Query-2	36%
Query-3	14%
Query-4	13%
Query-5	2%
Query-6	21%
Query-7	16%
Query-8	5%
Query-9	6%
Query-10	9%
Query-11	3%
Query-12	17%
Query-13	3%
Query-14	20%
Query-15	20%
Query-16	4%
Query-17	25%
Query-18	9%
Query-19	24%



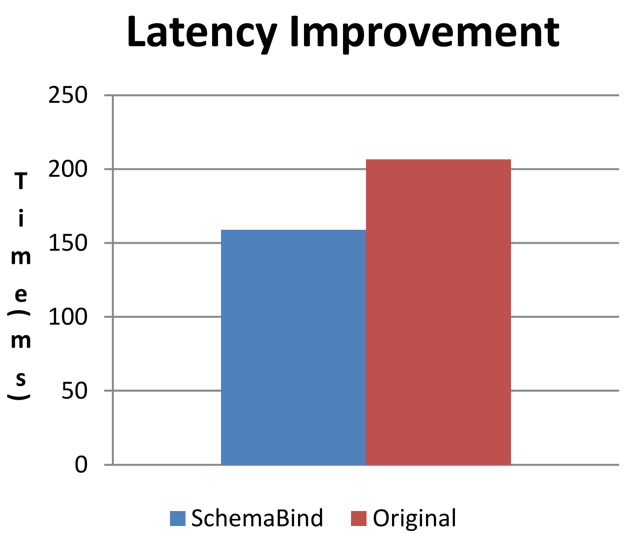
Query-1, 2 and 17 not shown in charts to maintain clear visibility of chart.

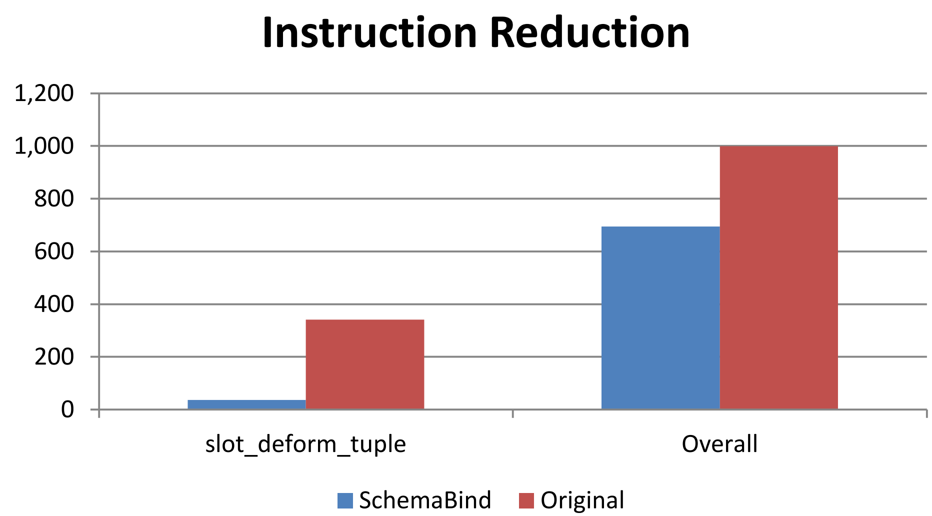
Performance (Hash Join):

Outer Table: Having 10 columns, cardinality 1M

Inner Table: Having 2 columns, cardinality 1K

Query: select sum(tbl.id10) from tbl,tbl2 where tbl.id10=tbl2.id2 group by tbl.id9;





Latency Improvement:	23%
Overall Instruction reduction:	30%
Access method instruction reduction:	89%

Performance Scenario:

Schema binding mainly depend on the code specialization of access function for table. Number of instruction reduced per call of slot_deform_function is more than 70% and hence if this function form good percentage of total instruction e.g. in

- Aggregate query,
- group
- Join
- Query with multiple attribute

All of above cases with huge table size, then overall instruction reduction will be also huge and hence much better performance.

Conclusion

Seeing the industry trend, we have implemented one way of code specialization, which resulted in up to 30% of performance improvement on standard benchmark TPC-H.

This technology will make us align with current business trend to tackle the CPU bottleneck and also could be one of the hot technology for work on PostgreSQL.

Acknowledgment

I would like to thanks my colleague ***Guogen Zhang, Yonghua Ding and Chen Zhu*** *who supported during this work.*

Reference

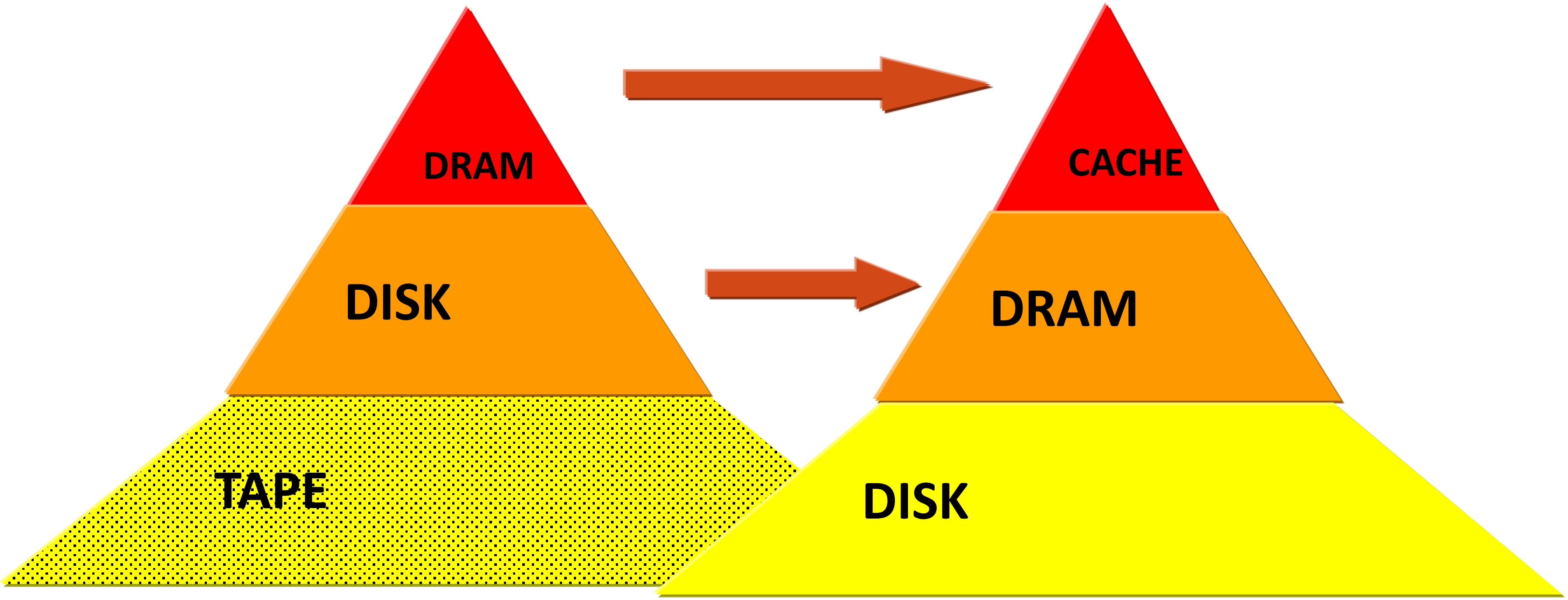
1. Zhang, Rui, Saumya Debray, and Richard T. Snodgrass. "Micro-specialization: dynamic code specialization of database management systems." *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012.

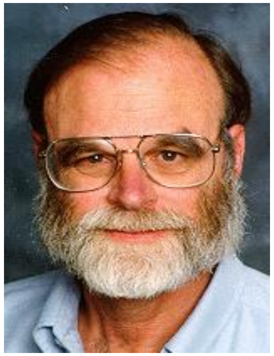
   http://dl.acm.org/citation.cfm?id=2259025

2. Freedman, Craig, Erik Ismert, and Per-Åke Larson. "Compilation in the Microsoft SQL Server Hekaton Engine." *IEEE Data Eng. Bull.* 37.1 (2014): 22-30.

   http://www.internalrequests.org/showconfirmpage/?url=ftp://131.107.65.22/pub/debull/A14mar/p22.pdf

PostgreSQL on Big RAM





“Disk is the new tape;
Memory is the new disk.”

-- Jim Gray

Source: ICDE Conference


THANK
YOU