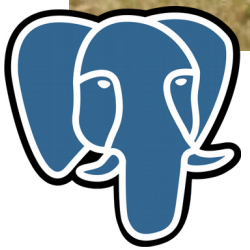
The path of a Query

PostgreSQL internals @ SFSCON 2015



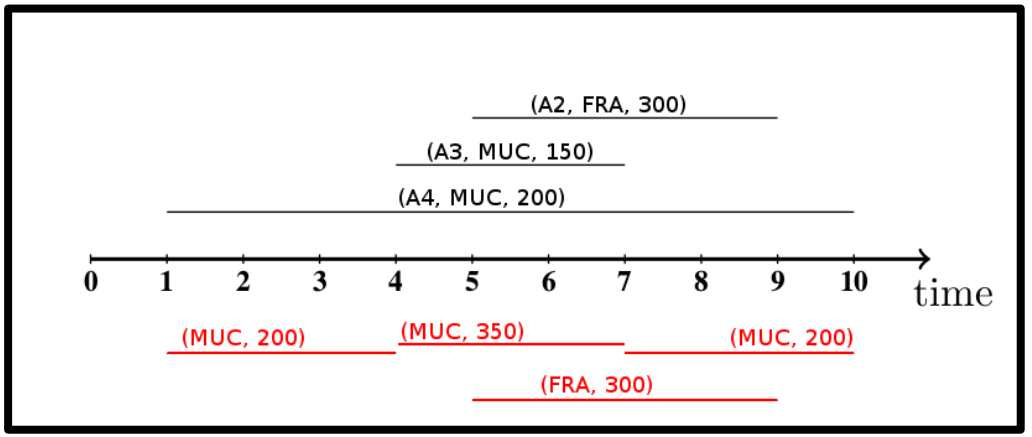
A glance into the
PostgreSQL kernel

by Peter Moser,
master student @ unibz

as an
intern @ TiS innovation park

Our Project: Temporal PostgreSQL

- Temporal queries missing

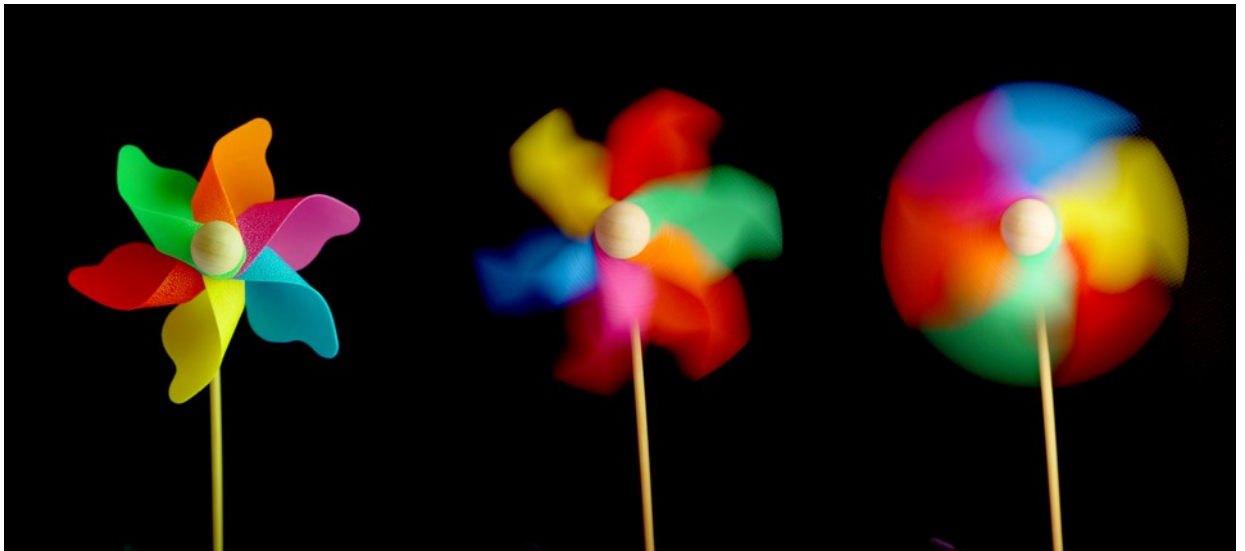


Temporal Aggregation Example

- Ongoing research at the Free University of BZ
- Extending the PostgreSQL kernel with temporal operators

Why creating patches for PostgreSQL?

- Extending SQL operators (not only functions)
- SQL standard compliance (missing features)
- Performance / Memory usage

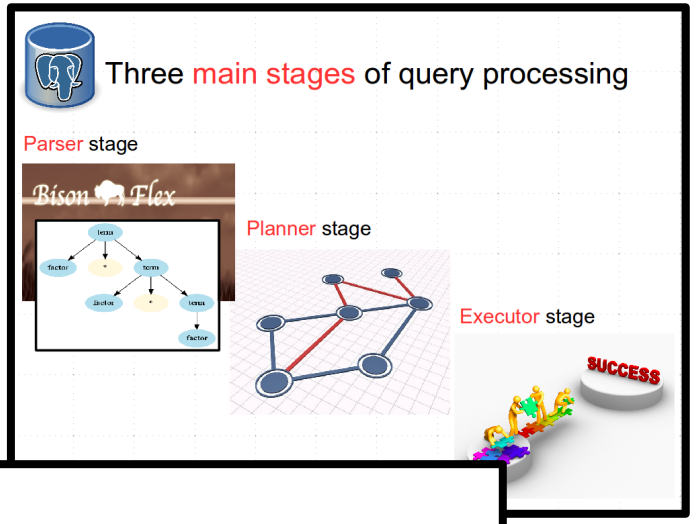


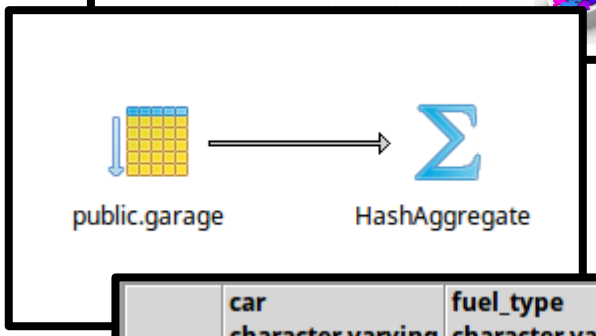
(c) Techlite

Talk (overview)

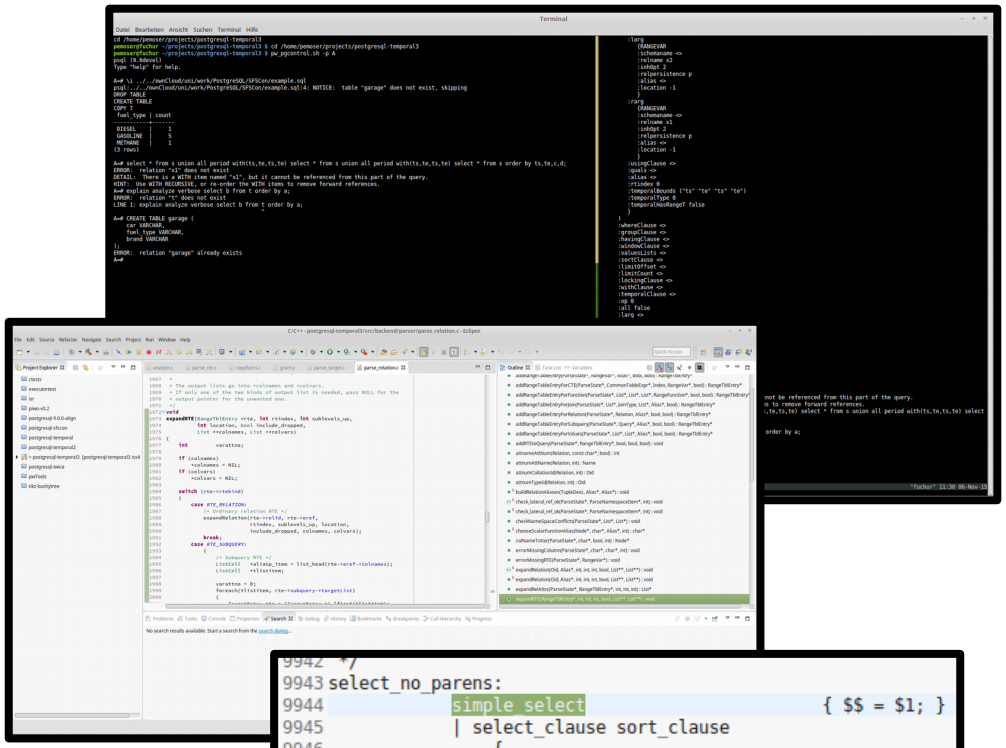
vs.

Workshop (code)





	car character varying	fuel_type character varying	brand character varying
1	T4	DIESEL	VW
2	Astra	GASOLINE	Opel
3	Golf	GASOLINE	VW
4	Corolla	GASOLINE	Toyota
5	Panda	METHANE	Fiat
6	Punto	GASOLINE	Fiat
7	500	GASOLINE	Fiat



```
9943 select_no_parens:
9944             simple_select                          { $$ = $1; }
9945           | select_clause sort_clause
9946               {
9947                   insertSelectOptions((SelectStmt *) $1, $2
9948                                       NULL, NULL, NULL,
9949                                       yyscanner);
9950                   $$ = $1;
9951               }
9952           | select_clause opt_sort_clause for_locking_claus
9953               {
9954                   insertSelectOptions((SelectStmt *) $1, $2
9955                                       list_nth($4, 0), list
9956                                       NULL,
9957                                       yyscanner);
9958                   $$ = $1;
9959               }
9960           | select_clause opt_sort_clause select_limit opt_
9961               {
9962                   insertSelectOptions((SelectStmt *) $1, $2
9963                                       list_nth($3, 0), list
```

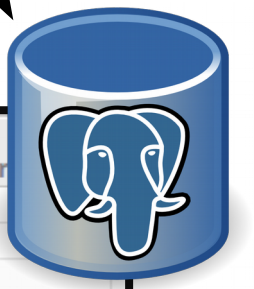
How does Postgres process a Query?

How many cars with a specific fuel type are in our garage?

WHAT? (Query)

```
SELECT
  fuel_type, COUNT(*)
FROM garage
GROUP BY fuel_type;
```


HOW?



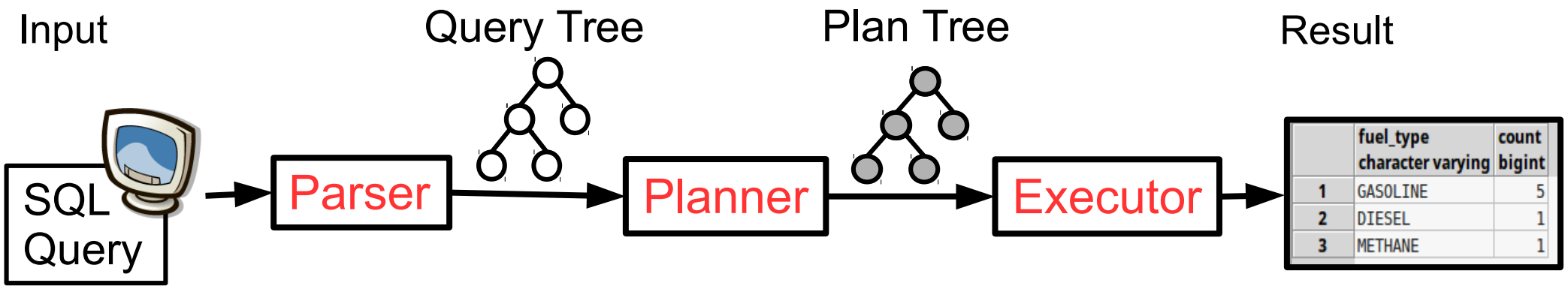
DATA

	car character varying	fuel_type character varying	brand character
1	T4	DIESEL	VW
2	Astra	GASOLINE	Opel
3	Golf	GASOLINE	VW
4	Corolla	GASOLINE	Toyota
5	Panda	METHANE	Fiat
6	Punto	GASOLINE	Fiat
7	500	GASOLINE	Fiat

Result

	fuel_type character varying	count bigint
1	GASOLINE	5
2	DIESEL	1
3	METHANE	1

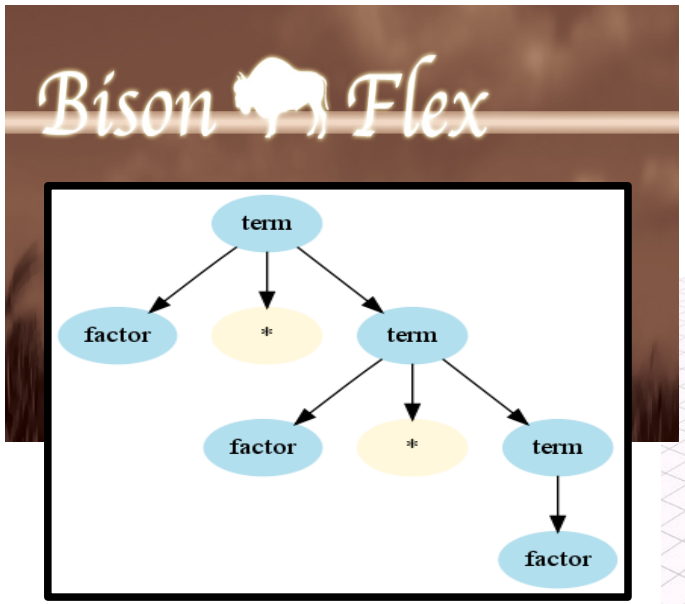
The path of a query from bird's-eye view



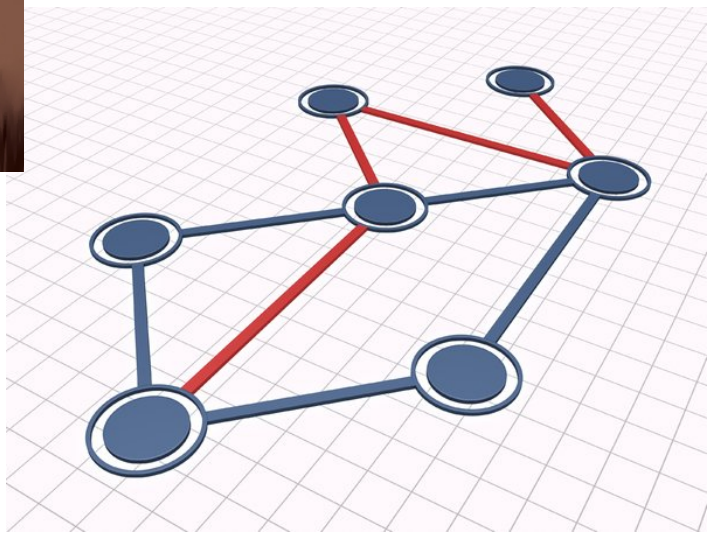
	fuel_type character varying	count bigint
1	GASOLINE	5
2	DIESEL	1
3	METHANE	1

Three main stages of query processing

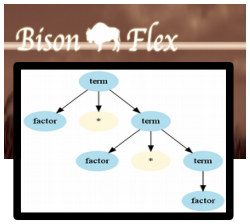
Parser stage



Planner stage



Executor stage



Parser stage

- Checks the SQL syntax
  - Syntax error:

```
SELECT * FRAM garage;
```

"FRAM" is not a SQL key word

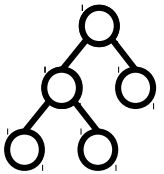
- Makes lookups in a catalog
  - Error in semantics:

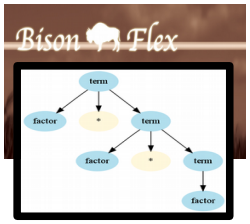
```
SELECT * FROM garauge;
```

  - Adds information (eg. column names, aliases)

Table "garauge" does not exist

- Builds the Query Tree
  - Internal representation of a Query





Parser stage: Identify tokens

Query String

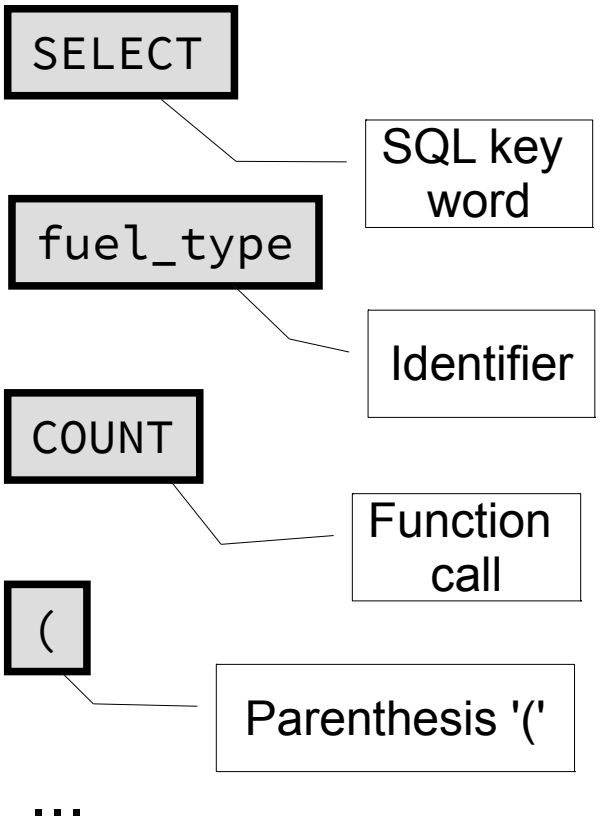
arrives as plain text

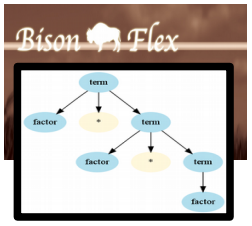
```
SELECT
  fuel_type, COUNT(*)
FROM garage
GROUP BY fuel_type;
```

(C) wiseGeek

The parser

generates tokens





Parser stage: Build query tree

Grammar rules

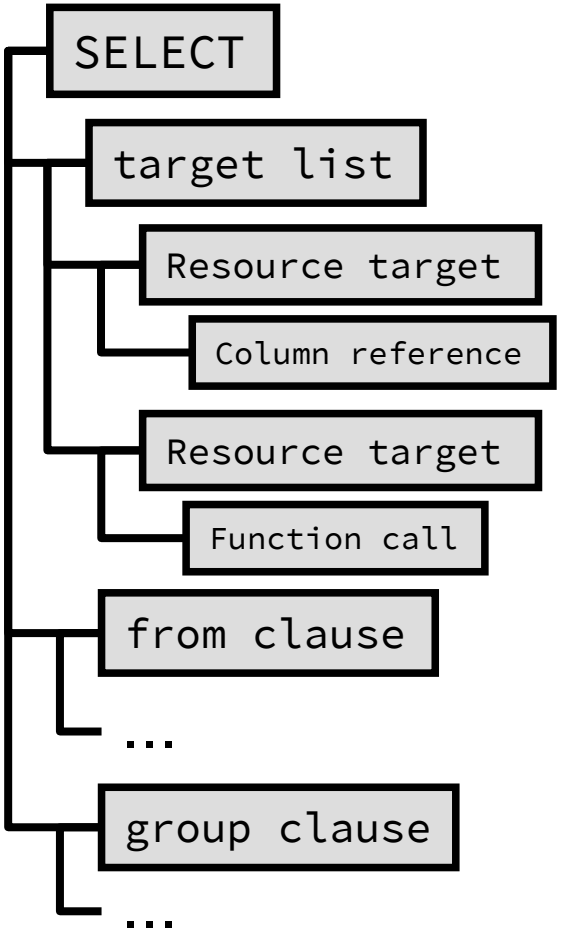
```
simple_select:
        SELECT target_list
        FROM from_clause
        GROUP BY group_clause
```

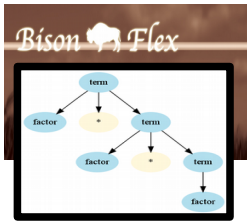
```
target_list:
        target_el
        | target_list ',' target_el
```

```
target_el:
        a_expr AS ColLabel
        | a_expr
        | '*'
```

```
a_expr:
        ...
```

Query tree





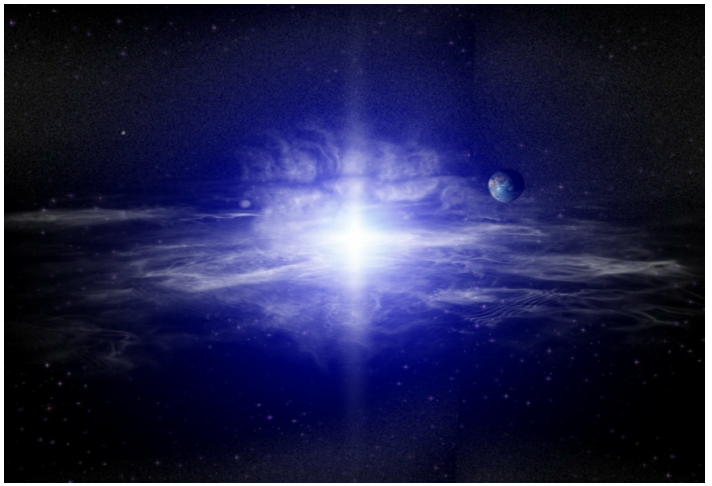
Query Tree Annotation

- Example: A_STAR expansion

```
SELECT   *   FROM garage;
```

```
SELECT car, fuel_type, brand FROM garage;
```

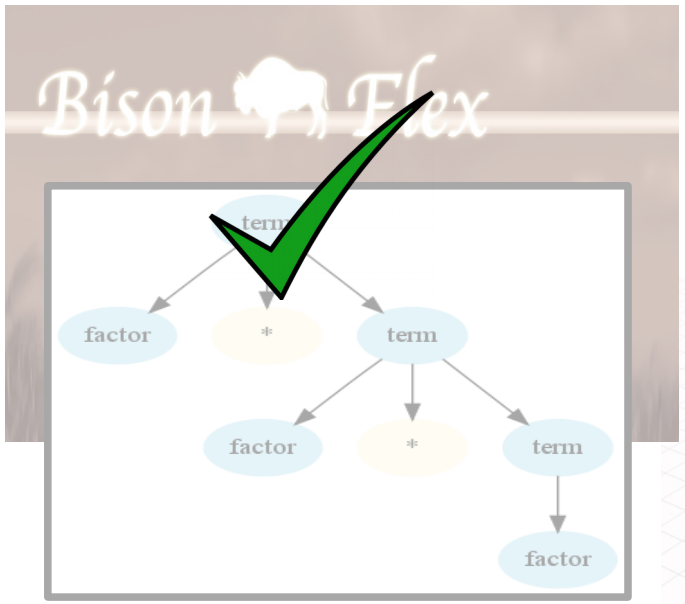
Data type, alias, ...



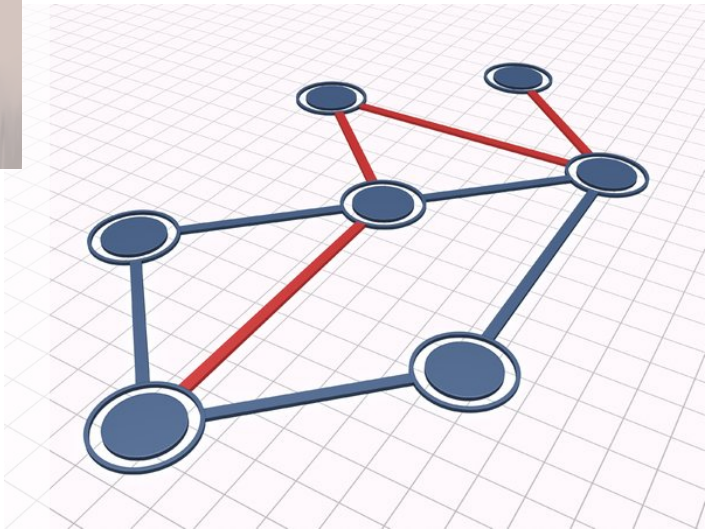
(C) wallpaperhi

Three main stages of query processing

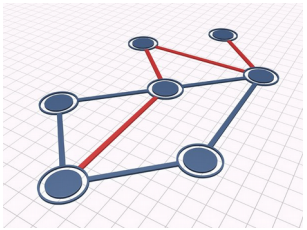
Parser stage



Planner stage

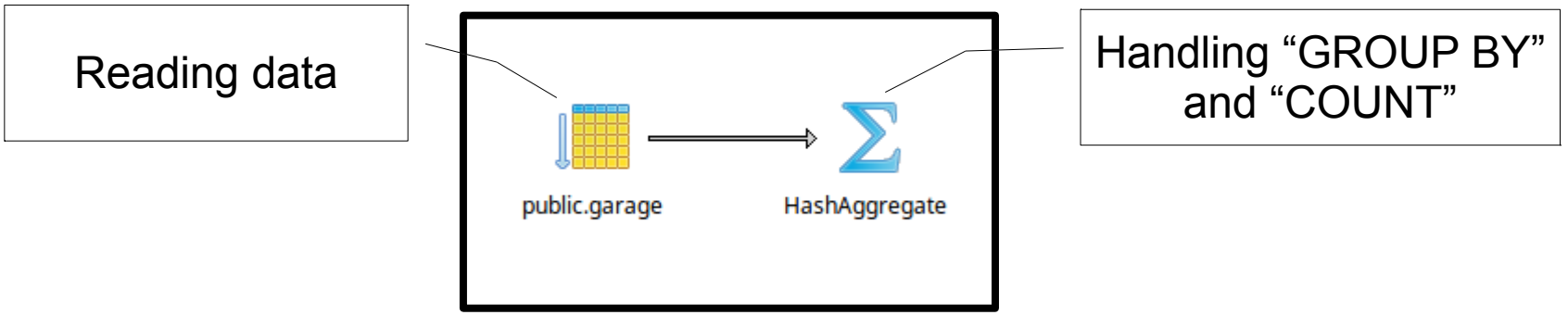


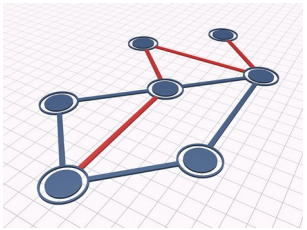
Executor stage



Planner stage

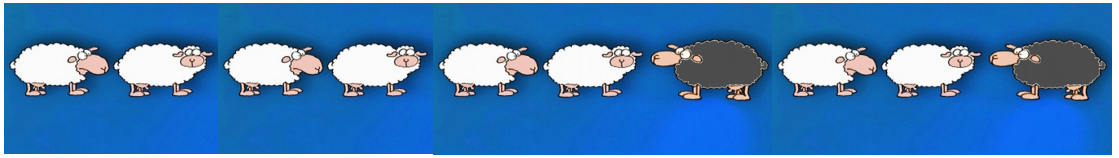
- There is a standard way to execute SQL
- The planner tries to find the best plan tree
- Creates the Plan Tree





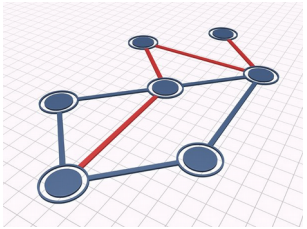
Planner decisions

- Access Scan Types
  - Sequential Scan, (Bitmap) Index Scan, Index-Only Scan



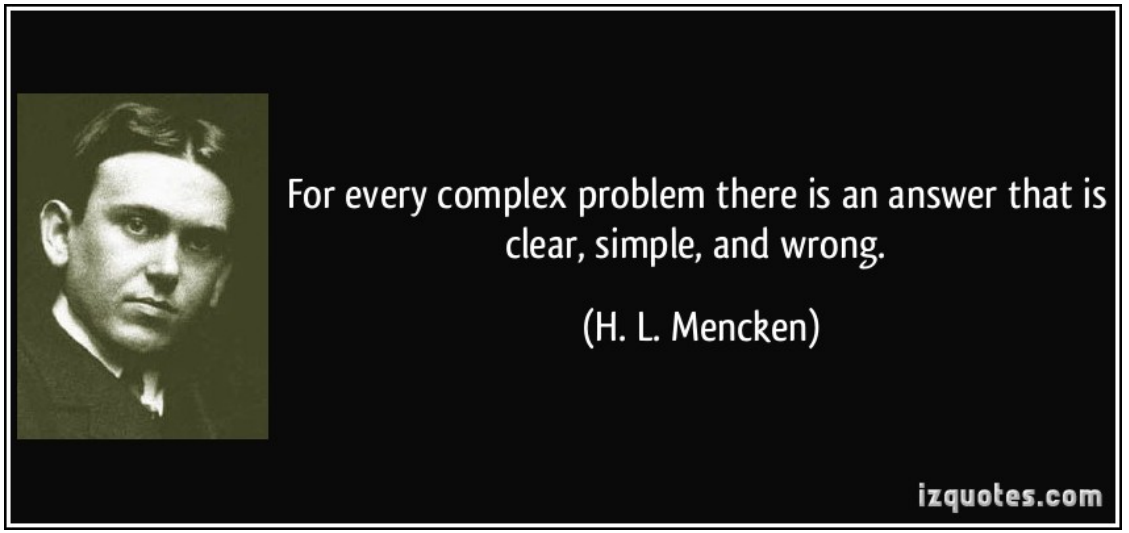
- Join
  - Join order
  - Join strategy: nested loop, merge join, hash join
  - Inner vs. Outer join
- Aggregation
  - Plain, sorted, hashed

MIN, MAX, COUNT, AVG, SUM, ...

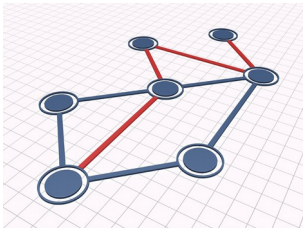


Planner stage

- Query planning is a complex problem
- Finding all plans could be too expensive
- If it is not feasible, it uses heuristics



Wikimedia Commons

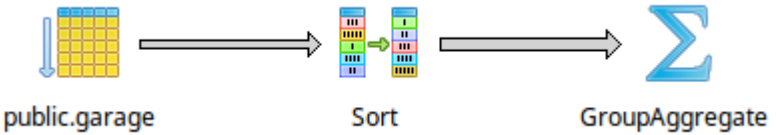


Planner example

```
SELECT fuel_type, COUNT(*)
FROM garage
GROUP BY fuel_type;
```

50

Plan #1

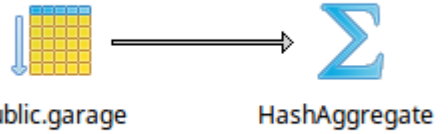


Cost estimates

- Scan: 16
- Sort: 48
- Group: **50**

20

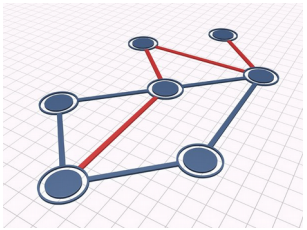
Plan #2



Cost estimates

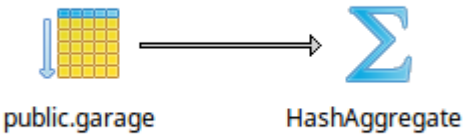
- Scan: 16
- HashAgg: **20**

Smaller cost means faster!



is Plan #2

```
SELECT fuel_type, COUNT(*)
FROM garage
GROUP BY fuel_type;
```



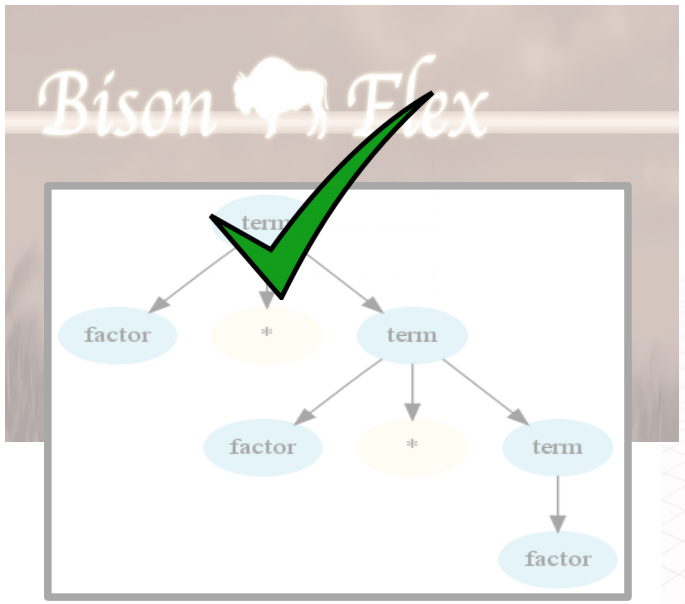
FINAL CHOICE:
Sequential Scan
on “public.garage”
and
aggregation with
a hash

Explaining the Query Plan Tree in pgAdmin

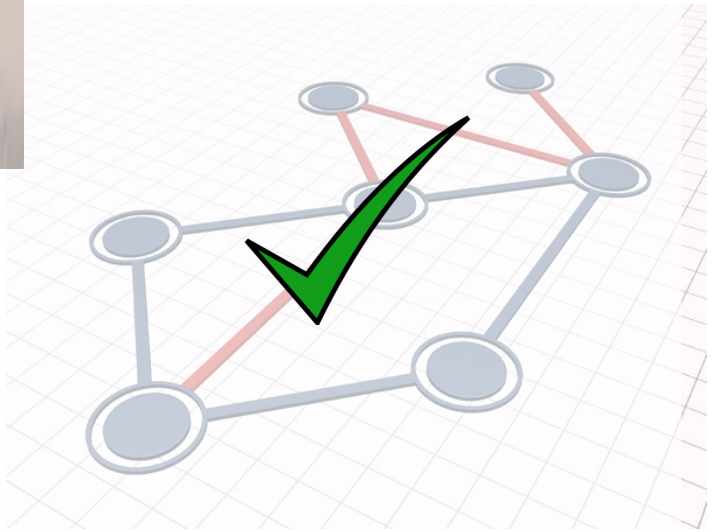
	QUERY PLAN text
1	HashAggregate (cost=19.75..21.75 rows=200 width=32) (actual time=0.024..0.025 rows=3 loops=1)
2	Group Key: fuel_type
3	-> Seq Scan on garage (cost=0.00..16.50 rows=650 width=32) (actual time=0.010..0.012 rows=7 loops=1)
4	Planning time: 0.071 ms
5	Execution time: 0.075 ms

Three main stages of query processing

Parser stage



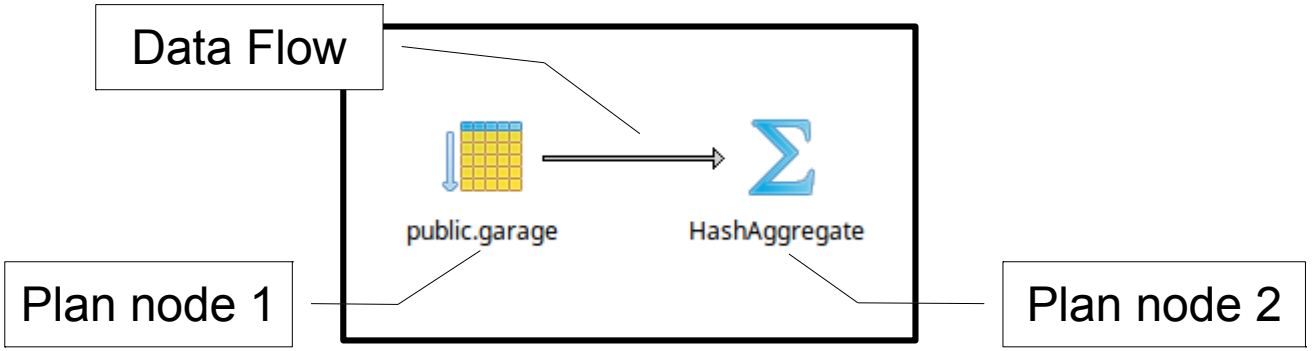
Planner stage



Executor stage

Executor stage

- Processes the (fastest) Plan Tree



- Each plan node returns a single row on demand

- Creates the Result

	fuel_type character varying	count bigint
1	GASOLINE	5
2	DIESEL	1
3	METHANE	1

Executor implementation

- Each executor node has a strict interface

`ExecInitNode()` `ExecProcNode()` ... `ExecEndNode()`

- Interface routines:

```
ExecInitNode():
    initialize a plan node and its
        subplans

ExecProcNode():
    get a tuple by executing
        the plan node
    return the tuple, or NULL

ExecEndNode():
    shut down a plan node and its
        subplans
```

Calling "ExecProcNode" twice

	car character varying	fuel_type character varying
1	T4	DIESEL
2	Astra	GASOLINE
3	Golf	GASOLINE
4	Corolla	GASOLINE
5	Panda	METHANE
6	Punto	GASOLINE
7	500	GASOLINE

Executor stage by example

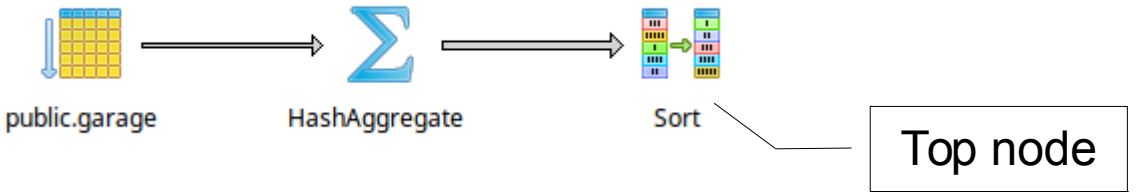
- **Our query** (+ ordering):

```
SELECT fuel_type, COUNT(*)
FROM garage
GROUP BY fuel_type
ORDER BY count DESC;
```

Data:

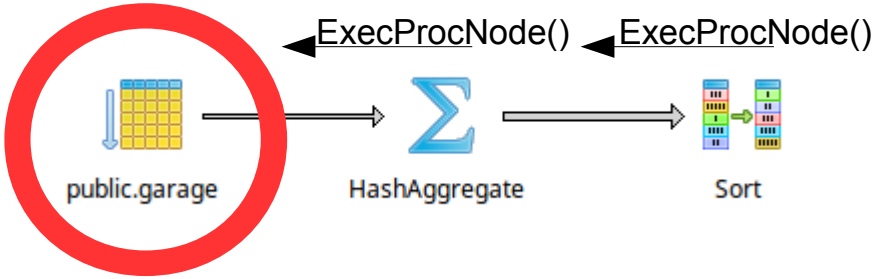
	car character varying	fuel_type character varying	brand character varying
1	T4	DIESEL	VW
2	Astra	GASOLINE	Opel
3	Golf	GASOLINE	VW
4	Corolla	GASOLINE	Toyota
5	Panda	METHANE	Fiat
6	Punto	GASOLINE	Fiat
7	500	GASOLINE	Fiat

- Execution Plan:



- The executor recursively calls itself to process subplans (starting from the top node Sort)

Executor stage




```
SELECT fuel_type, COUNT(*)
FROM garage
GROUP BY fuel_type
ORDER BY count DESC;
```

- Run the Sequential Scan node
- Return the result to the node HashAggregate
- Input:

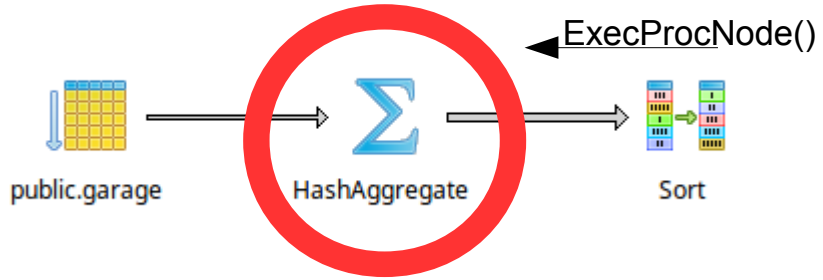
(c) Simutek

→

Result:

	car character varying	fuel_type character varying	brand character varying
1	T4	DIESEL	VW
2	Astra	GASOLINE	Opel
3	Golf	GASOLINE	VW
4	Corolla	GASOLINE	Toyota
5	Panda	METHANE	Fiat
6	Punto	GASOLINE	Fiat
7	500	GASOLINE	Fiat

Executor stage




```
SELECT fuel_type, COUNT(*)
FROM garage
GROUP BY fuel_type
ORDER BY count DESC;
```

- Build a hash upon “fuel_type”
- Use the hash to find candidates for fuel type groups
- Grouping by hash

Group #1
Group #2
Group #3

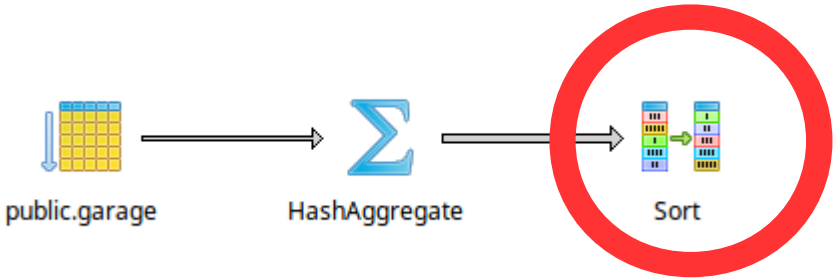
	car character varying	fuel_type character varying	brand character varying	hash text
1	T4	DIESEL	VW	167
2	Panda	METHANE	Fiat	b01
3	Golf	GASOLINE	VW	e6c
4	Punto	GASOLINE	Fiat	e6c
5	500	GASOLINE	Fiat	e6c
6	Corolla	GASOLINE	Toyota	e6c
7	Astra	GASOLINE	Opel	e6c

→

Result

	fuel_type character varying	count bigint
1	DIESEL	1
2	GASOLINE	5
3	METHANE	1

Executor stage




```
SELECT fuel_type, COUNT(*)
FROM garage
GROUP BY fuel_type
ORDER BY count DESC;
```

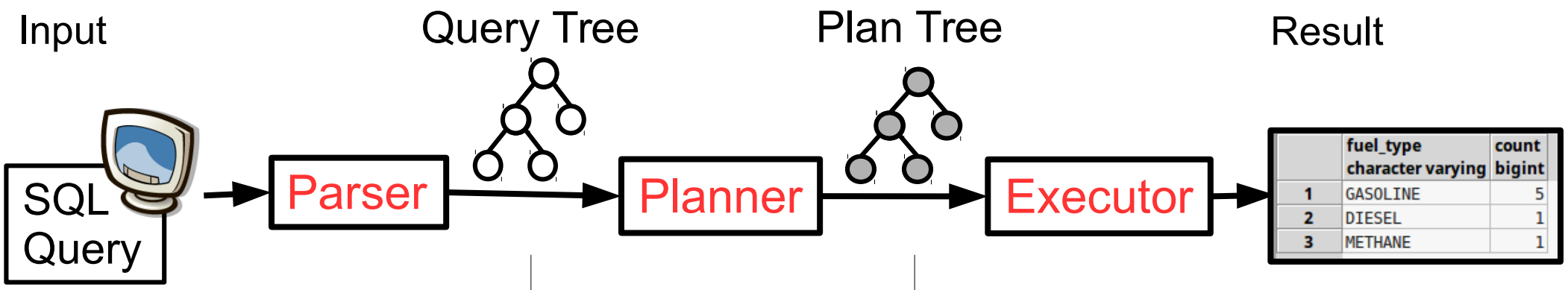
- Sort the rows from the subplan HashAggregate with sort key “count”
- Return the sorted groups as result

	fuel_type character varying	count bigint
1	DIESEL	1
2	GASOLINE	5
3	METHANE	1

→

	fuel_type character varying	count bigint
1	GASOLINE	5
2	DIESEL	1
3	METHANE	1

In summary, the path of a query...



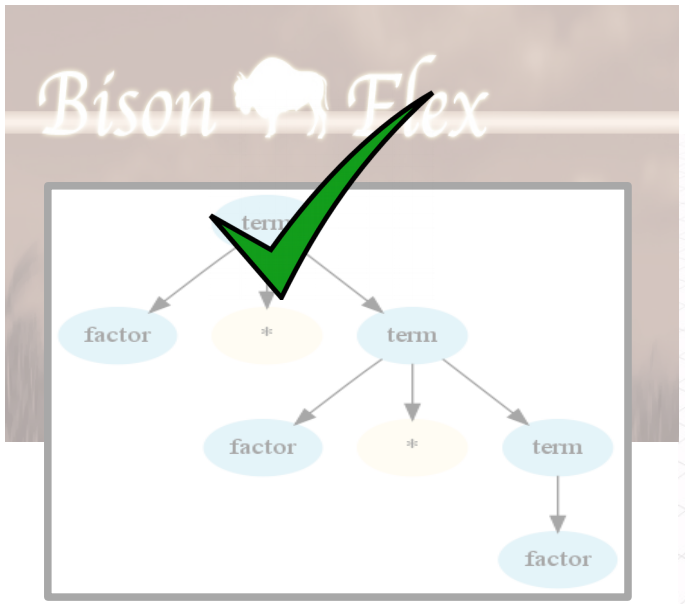
- checks SQL syntax
- Builds the Query Tree

- Finds a good Execution Plan
- Builds the Plan Tree

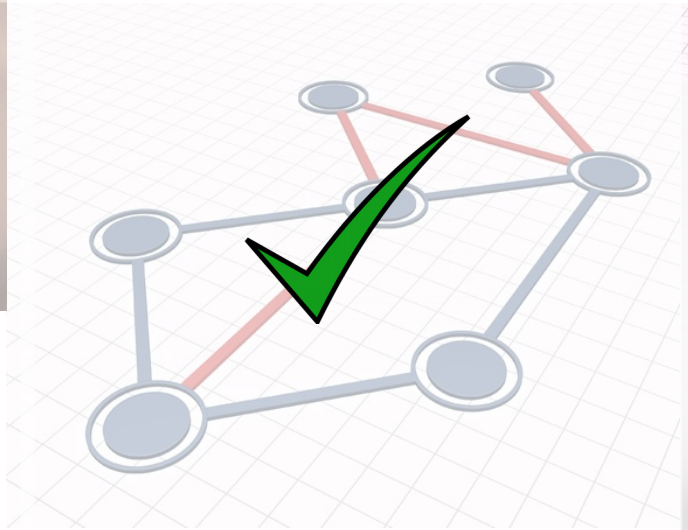
- Executes the Plan
- Builds the Result

Three main stages of query processing

Parser stage



Planner stage



Executor stage

Thank you!

Additional information

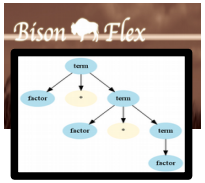
Workshop Tutorial @ github

https://github.com/Piiit/postgres-twice-patch/blob/master/SFSCON15-README.md

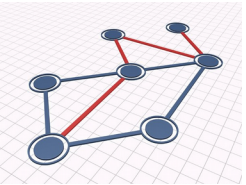
...or...

http://bit.ly/1OyU0tD

In summary, the path of a query...



- Parser
  - checks SQL syntax
  - builds a query tree



- Planner / Optimizer
  - Finds an (not always) optimal plan
  - Tests all possibilities or uses heuristics

- Executor
  - Executes the plan
  - Is implemented as an iterator

Three main stages of query processing

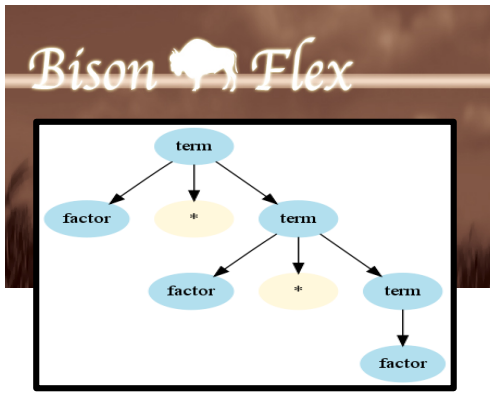
SQL Query

```
SELECT fuel_type, COUNT(*)
FROM garage
GROUP BY fuel_type;
```

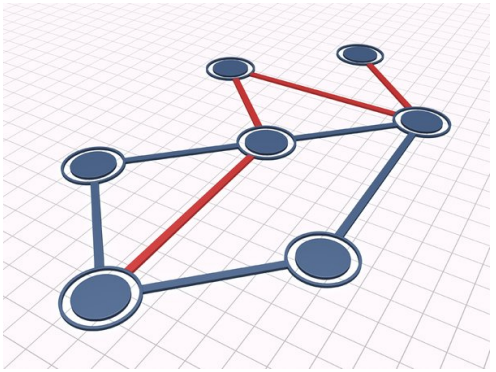
Result

	fuel_type character varying	count bigint
1	DIESEL	1
2	GASOLINE	5
3	METHANE	1

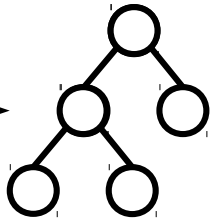
Parser stage



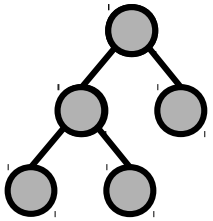
Planner stage



Executor stage

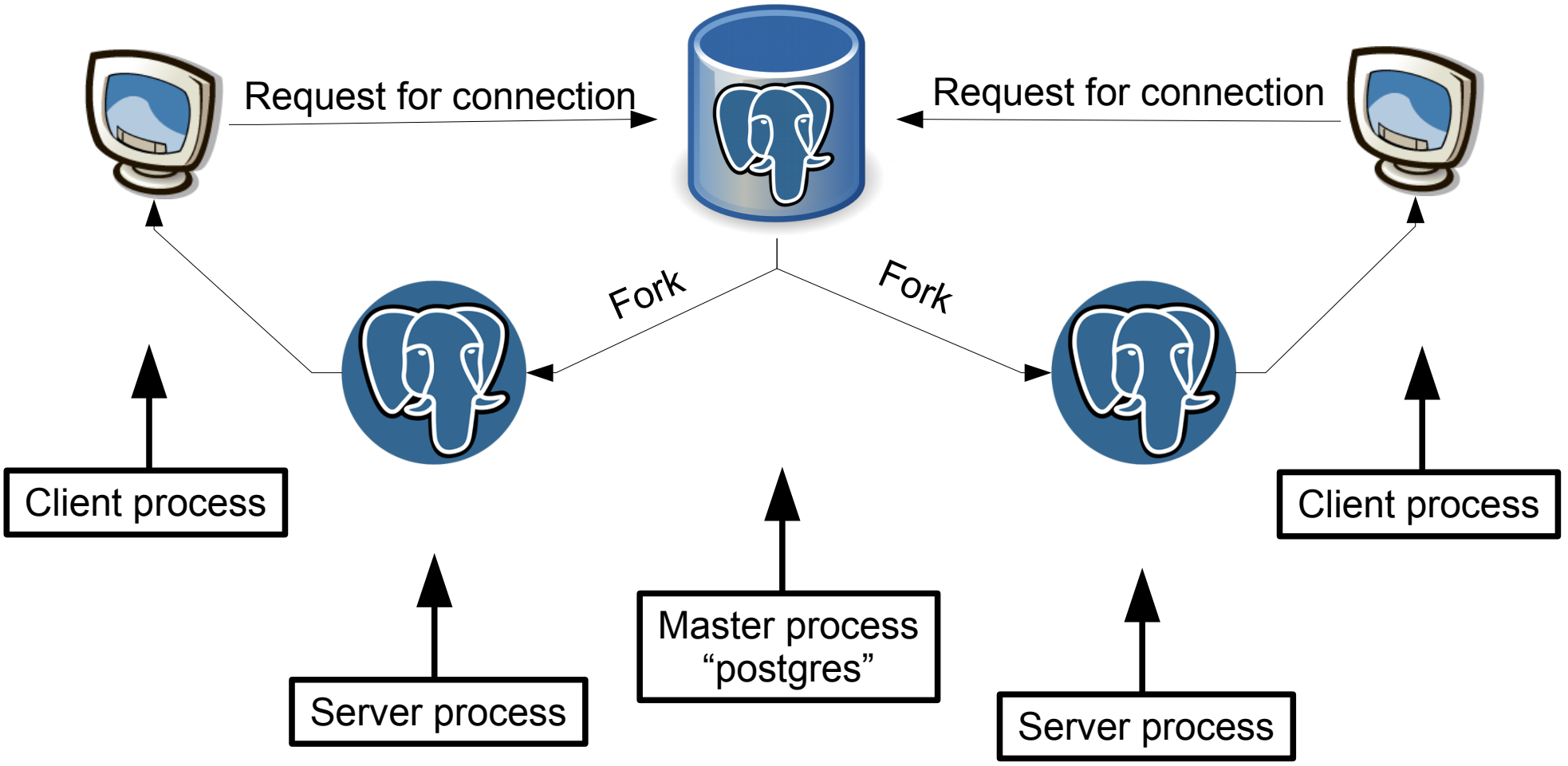


Query Tree



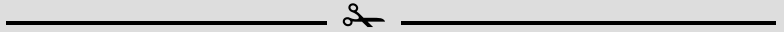
Plan Tree

PostgreSQL Server / Client interaction



Simplified Parse Tree

```
{SELECT
:distinctClause <>
:intoClause <>
:targetList (
   {RESTARGET
   :name <>
   :indirection <>
   :val
      {COLUMNREF
      :fields ("car")
      :location 7
      }
   }
   {RESTARGET
   :name <>
   :indirection <>
   :val
      {FUNCCALL
      :funcname ("count")
      :args <>
      :agg_star true
      :over <>
      :location 12
      }
   }
)
```



```
   :fromClause (
      {RANGEVAR
      :relname garage
      :alias <>
      :location 26
      }
   )
   :whereClause <>
   :groupClause (
      {COLUMNREF
      :fields ("car")
      :location 42
      }
   )
   :havingClause <>
   :windowClause <>
   :valuesLists <>
   :sortClause <>
   :limitCount <>
   :lockingClause <>
   :withClause <>
   :op 0
   :all false
   :larg <>
   :rarg <>
   }
```

pretty-print output of pprint()