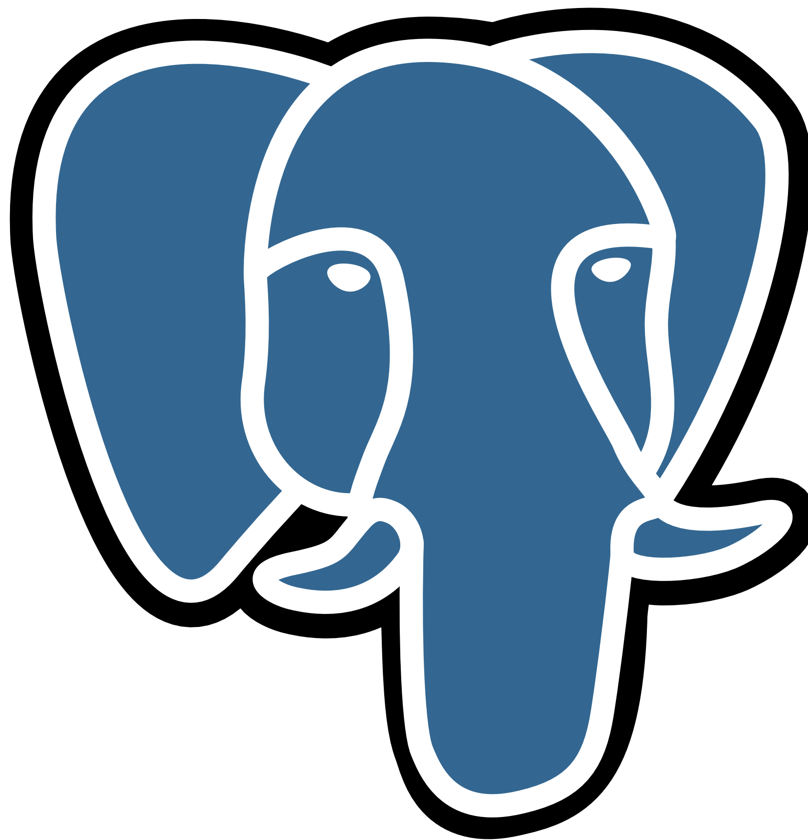
A Tour of PostgreSQL Internals



Tom Lane
Great Bridge, LLC
tgl@sss.pgh.pa.us

Outline

I'm going to present three separate views of PostgreSQL.

Each view is equally valid but will teach you something different about the beast.

- **VIEW 1: Processes and interprocess communication structure**

  Key ideas: client/server separation, inter-server communication

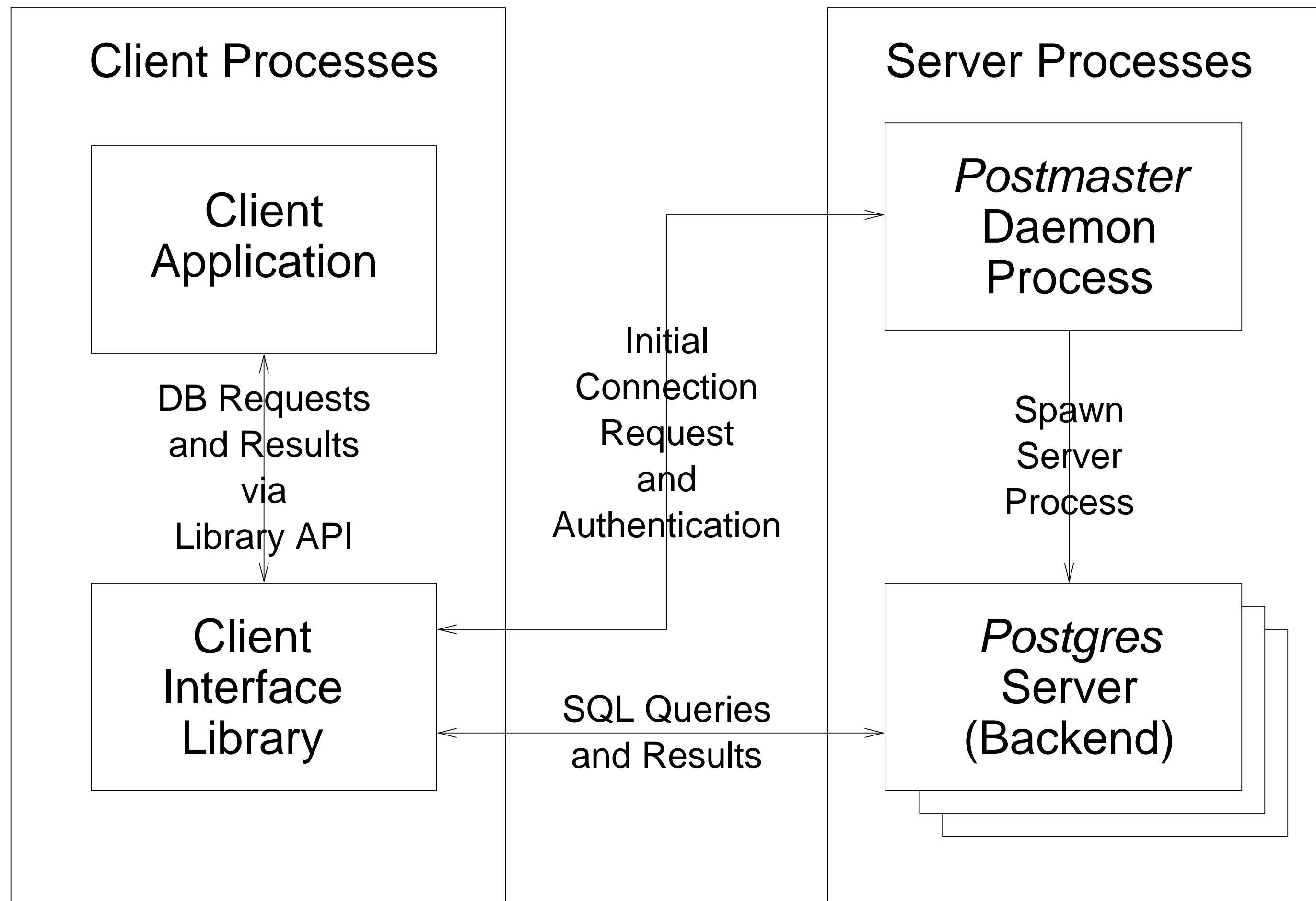
- **VIEW 2: System catalogs and data types**

  Key ideas: extensibility, compartmentalization

- **VIEW 3: Steps of query processing**

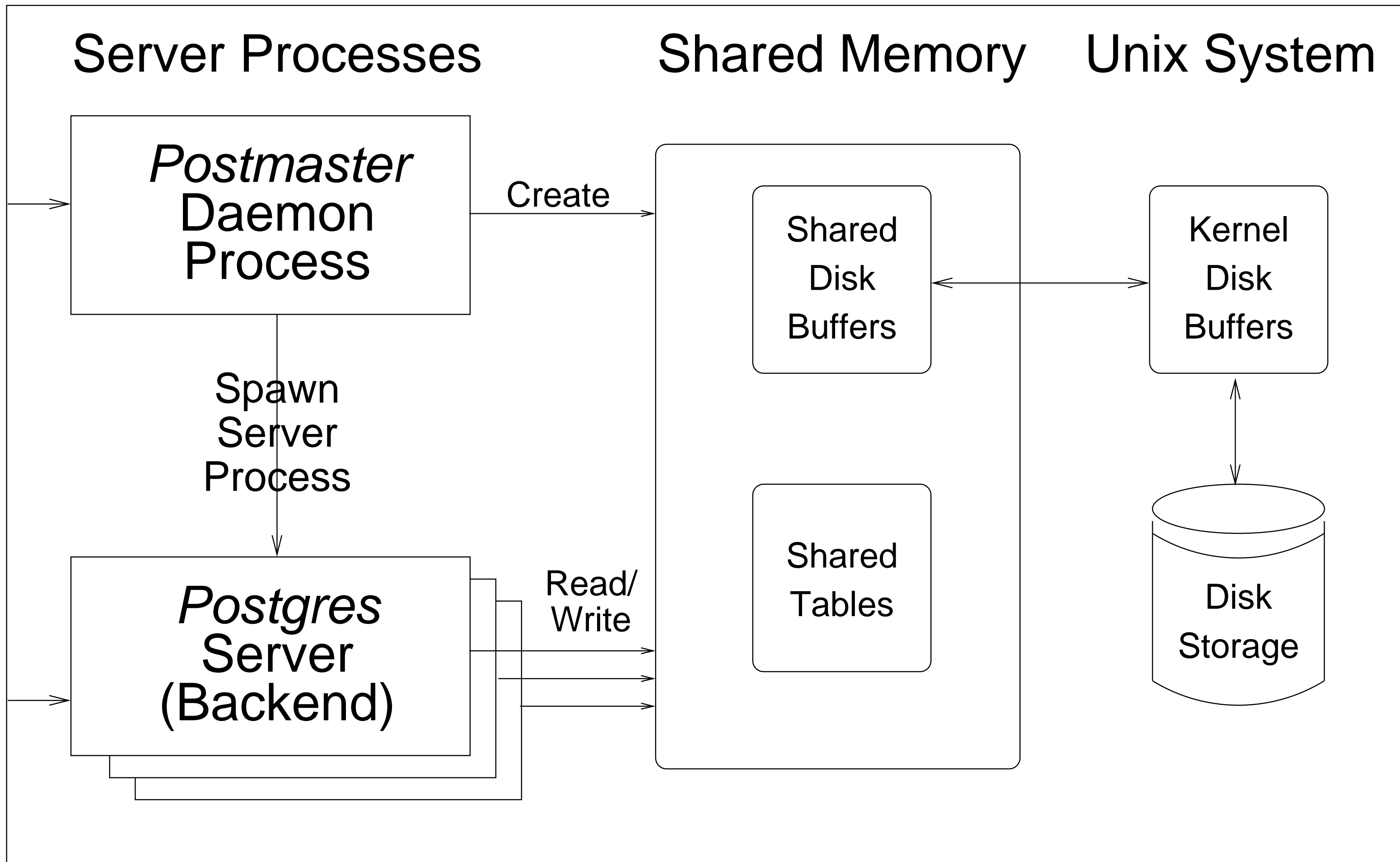
  Key ideas: parse and plan trees, scans and joins, cost estimation

PostgreSQL processes: client/server communication



- Multiple client libraries offer different APIs: libpq, ODBC, JDBC, Perl DBD
- Client libraries insulate client applications from changes in on-the-wire protocol

PostgreSQL processes: inter-server communication



PostgreSQL processes: summary

Pros:

- **Hard separation of client and server is good for security/reliability**
- **Works well in a networked environment**
- **Portable across most flavors of Unix**

Cons:

- **Dependence on shared memory for inter-server communication limits scalability**

  A single server site can’t be spread across multiple machines

- **Connection startup overhead is bad for short-duration client tasks**

  Usual workaround: client-side connection pooling, as in AOLServer for example

System catalogs and data types: introduction

Postgres is catalog-driven to a much greater extent than most other DBMSes.

- We have the usual sort of system catalogs to describe tables, their columns, their indexes, etc.

- But we also use system catalogs to store information about datatypes, functions, operators, index access methods, and so forth.

- The system can be extended by adding new catalog entries (and writing the underlying code, in the case of adding a function).

System catalogs and data types: basic catalogs

Basic system catalogs that describe a table:

pg_class: one row for each table in the database, containing name, owner, access permissions, number of columns, etc.

pg_attribute: one row for each column of each table, containing name, data type, column number, etc.

pg_index: relates a table to its indexes. (Indexes are tables and so have their own entries in **pg_class** and **pg_attribute,** too.) One row per index, containing references to **pg_class** entries of index and underlying table, plus info about which columns of the table the index is computed on and what the index operators are.

Lots of less-important tables, such as **pg_relcheck** which stores constraint definitions, but I'm just hitting the high spots.

System catalogs and data types: functions

pg_proc: defines functions. Each function has one row that gives its name, input argument types, result type, implementation language, and definition (either the text, if it's in an interpreted language, or a reference to its executable code, if it's compiled). Compiled functions can be statically linked into the server, or stored in shared libraries that are dynamically loaded on first use. Compiled functions are typically written in C, though in theory one could use other choices.

pg_language: defines implementation languages for functions. Languages with hard-wired support are **internal** (statically-linked compiled code), **C** (dynamically-linked compiled code), and **SQL** (body is one or more SQL queries). Optional languages currently include pl/pgsql (PL/SQL-ish), tcl, and perl, with more to come. These languages are supported by handlers that are dynamically-linked functions --- the core server doesn't know a thing about them.

System catalogs and data types: example functions

C:

```
int4
square_int4 (int4 x)
{
    return x * x;
}
```

Compile the above into a shared-library file, then say:

```
CREATE FUNCTION square(int4) RETURNS int4
AS '/path/to/square.so', 'square_int4'
LANGUAGE 'C';
```

PL/PGSQL:

```
CREATE FUNCTION square(int4) RETURNS int4 AS 'begin
return $1 * $1;
end;' LANGUAGE 'plpgsql';
```

System catalogs and data types: aggregate functions

pg_aggregate: defines aggregate functions like min(), max(), count(). Aggregates involve a working-state datatype, an update function, and a final-output function.

AVG(int4) in pltcl:

```
-- The working state is a 2-element integer array, sum and count.
-- We use split(n) as a quick-and-dirty way of parsing the input array
-- value, which comes in as a string like '{1,2}'.  There are better ways...

create function tcl_int4_accum(int4[], int4) returns int4[] as '
    set state [split $1 "{,}"]
    set newsum [expr {[lindex $state 1] + $2}]
    set newcnt [expr {[lindex $state 2] + 1}]
    return "{$newsum,$newcnt}"
' language 'pltcl';

create function tcl_int4_avg(int4[]) returns int4 as '
    set state [split $1 "{,}"]
    if {[lindex $state 2] == 0} { return_null }
    return [expr {[lindex $state 1] / [lindex $state 2]}]
' language 'pltcl';

create aggregate tcl_avg (
        basetype = int4,              -- input datatype
        sfunc = tcl_int4_accum,       -- update function name
        stype = int4[],               -- working-state datatype
        finalfunc = tcl_int4_avg,     -- final-output function name
        initcond = '{0,0}'            -- initial value of working state
    );
```

System catalogs and data types: operators

pg_operator: defines operators that can be used in expressions.

An operator is mainly just syntactic sugar for a function of one or two arguments.

Do-it-yourself exponentiation operator:

```
CREATE FUNCTION mypower(float8, float8) RETURNS float8 AS 'begin
return exp(ln($1) * $2);
end;' LANGUAGE 'plpgsql';

CREATE OPERATOR ** (
    procedure = mypower,
    leftarg = float8,
    rightarg = float8 );

SELECT 44 ** 2, 81 ** 0.5;
 ?column? | ?column?
----------+----------
     1936 |        9
(1 row)
```

System catalogs and data types: data types

pg_type: defines the basic data types that values stored in tables can have and that are accepted and returned by operators and functions.

New data types can be invented by making new entries in **pg_type.**

Example: say your application wants to store coordinates of points in 3-D space.

The built-in type Point won't do (it's only 2-D), so you build a 3-D point datatype.

At minimum, a data type must have two associated functions, which convert its internal representation to and from external textual form. Having written these functions, you can define the type to Postgres:

```
CREATE TYPE point3 (
    input = point3in,
    output = point3out,
    internallength = 24,        -- space for three float8's
    alignment = double );       -- ensure storage will be aligned properly
```

System catalogs and data types: data types

Typically one wants to do more with a datatype than just store and retrieve values, so the next step is to add functions and operators that do useful things with the datatype. For instance, distance between two points is a function you'd probably want to have for your 3-D point type. This is where things get tedious...

A new datatype can be made indexable by adding entries to **pg_amop** and **pg_amproc.** These entries tell the indexing machinery about a set of comparison operators and functions that behave in the way an index access method expects. For example, to be btree-indexable, entries must be provided for a datatype's **< <= = > >=** comparison operators, as well as a 3-way sort comparison function.

None of the standard index types are very suitable for 3-D points (R-trees are geometric, but only 2-D). In theory you could write a new index access method for 3-D R-trees and link it into the system with a new **pg_am** entry. No one has actually done something like that in recent memory, however.

System catalogs and data types: summary

Pros:

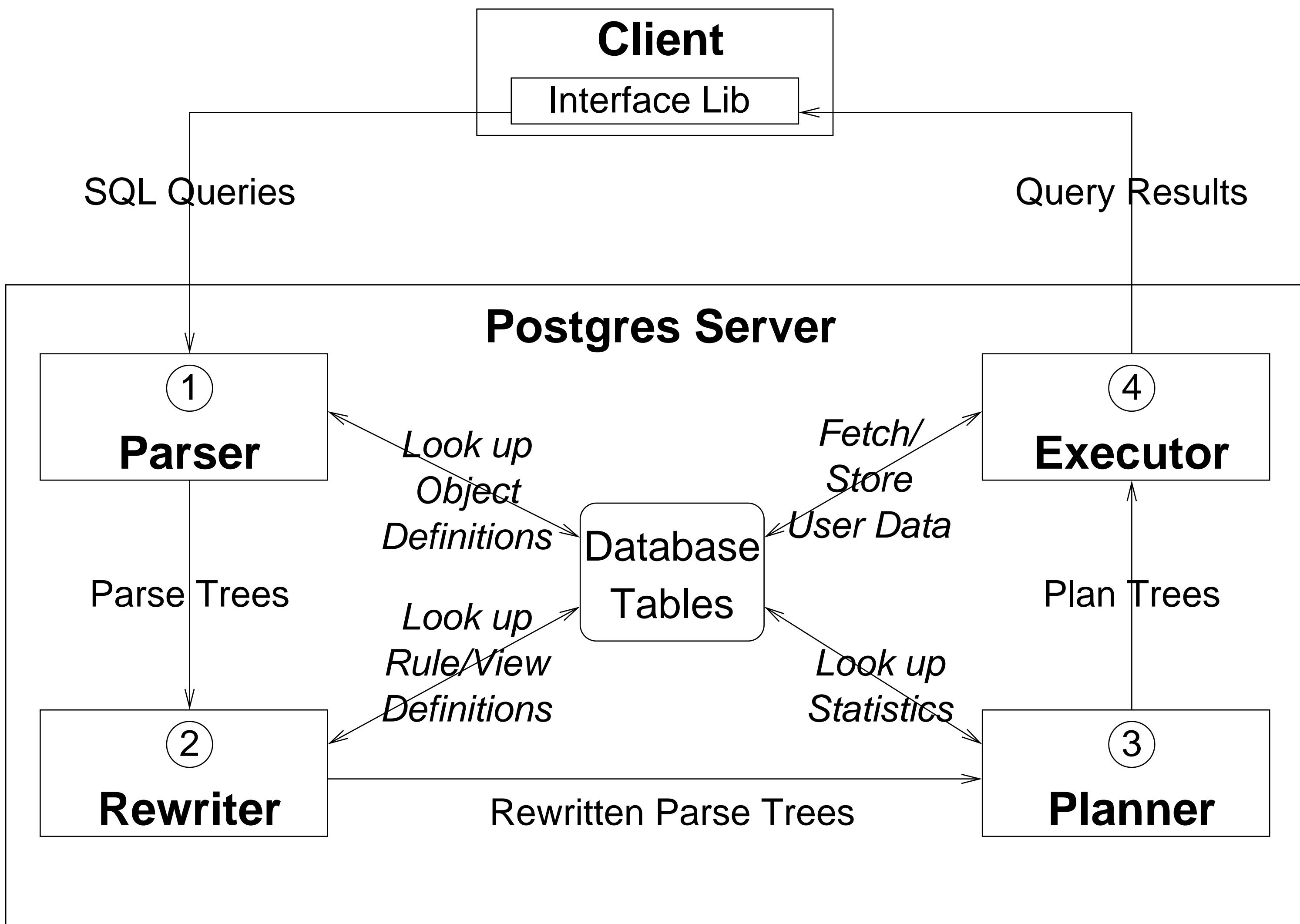
- **Extensibility of functions and datatypes is a big win for specialized applications**

Cons:

- **Making a new datatype with a usefully rich set of functions is a lot of work**
- **System design demands an "arm’s length", "black box" treatment of objects**

  For example, MAX() and MIN() aggregate functions are processed the same as all other aggregates: they must scan all the data. In the presence of an ordered index, these could be implemented more efficiently by looking at the extremal index value. But making this happen in a way that isn’t a complete kluge requires designing a general representation of a connection between aggregates and indexes, which is a rather hard problem. The system design isn’t very forgiving of partial solutions to issues like this.

Steps of query processing: overview



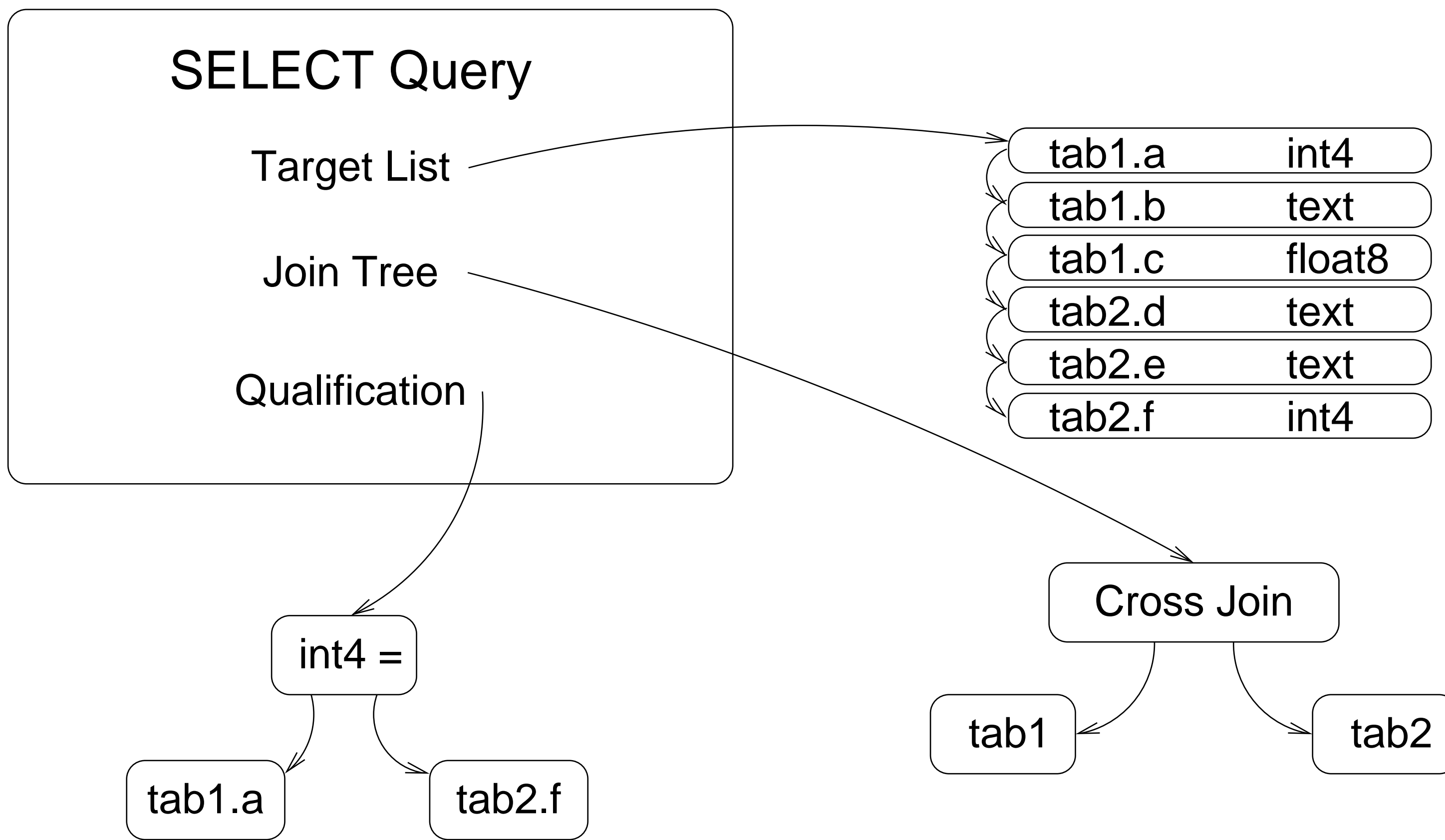
Key data structures: parse tree, plan tree

Steps of query processing: parser

Input:

```
SELECT * FROM tab1, tab2 WHERE tab1.a = tab2.f
```

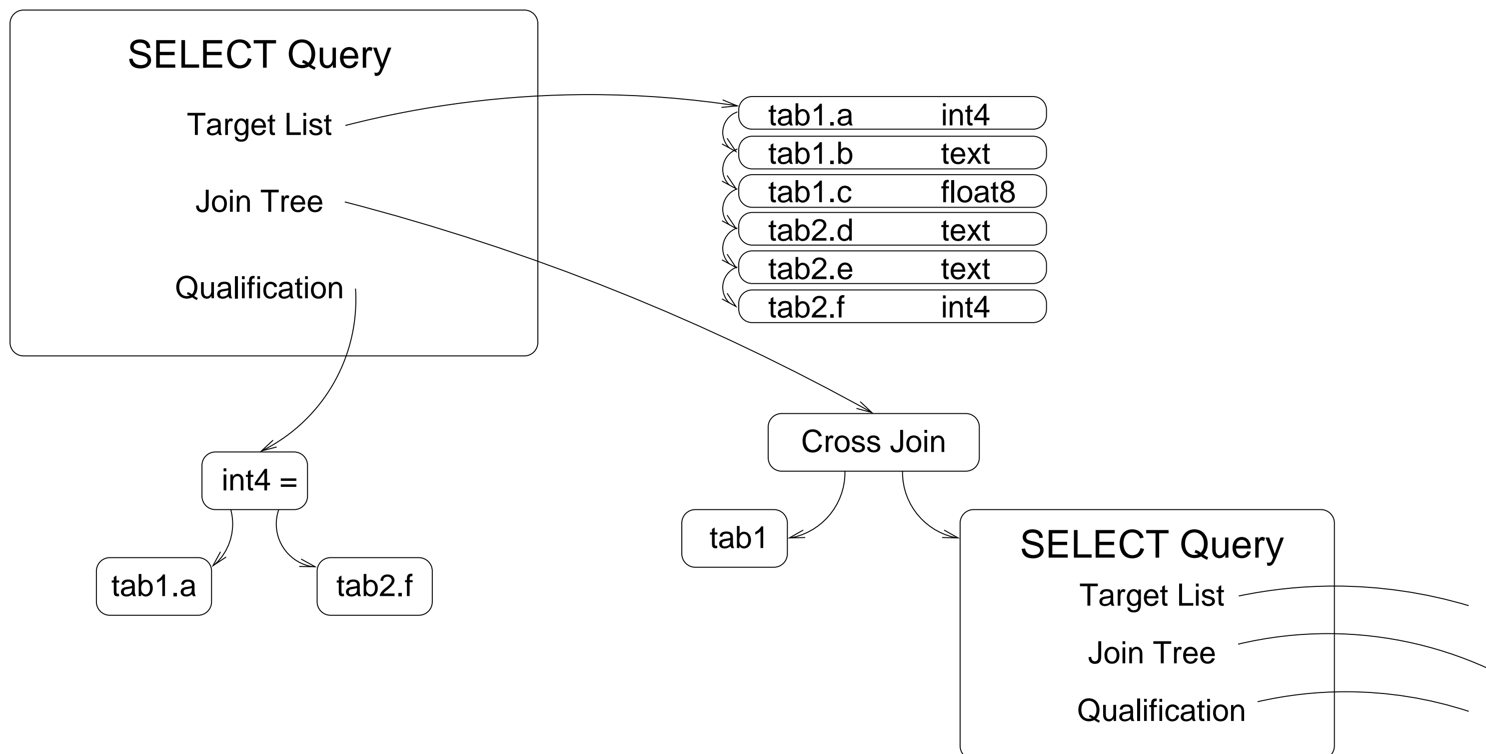
Output:



Steps of query processing: rewriter

Views are handled by substitution of subqueries into the parse tree.

For example, if tab2 were a view then the rewriter would emit something like this:



ON INSERT/UPDATE/DELETE rules require more extensive transformations, and may produce multiple queries from a single query.

Steps of query processing: executor

The basic idea of the executor is that it executes a plan tree, which is a pipelined demand-pull network of processing nodes. Each node produces the next tuple in its output sequence each time it is called. Upper-level nodes call their subnodes to get input tuples, from which they compute their own output tuples.

Bottom-level nodes are scans of physical tables --- either sequential scans or index scans.

Upper-level nodes are usually join nodes --- nested-loop, merge, and hash joins are available. Each join node combines two input tuple streams into one.

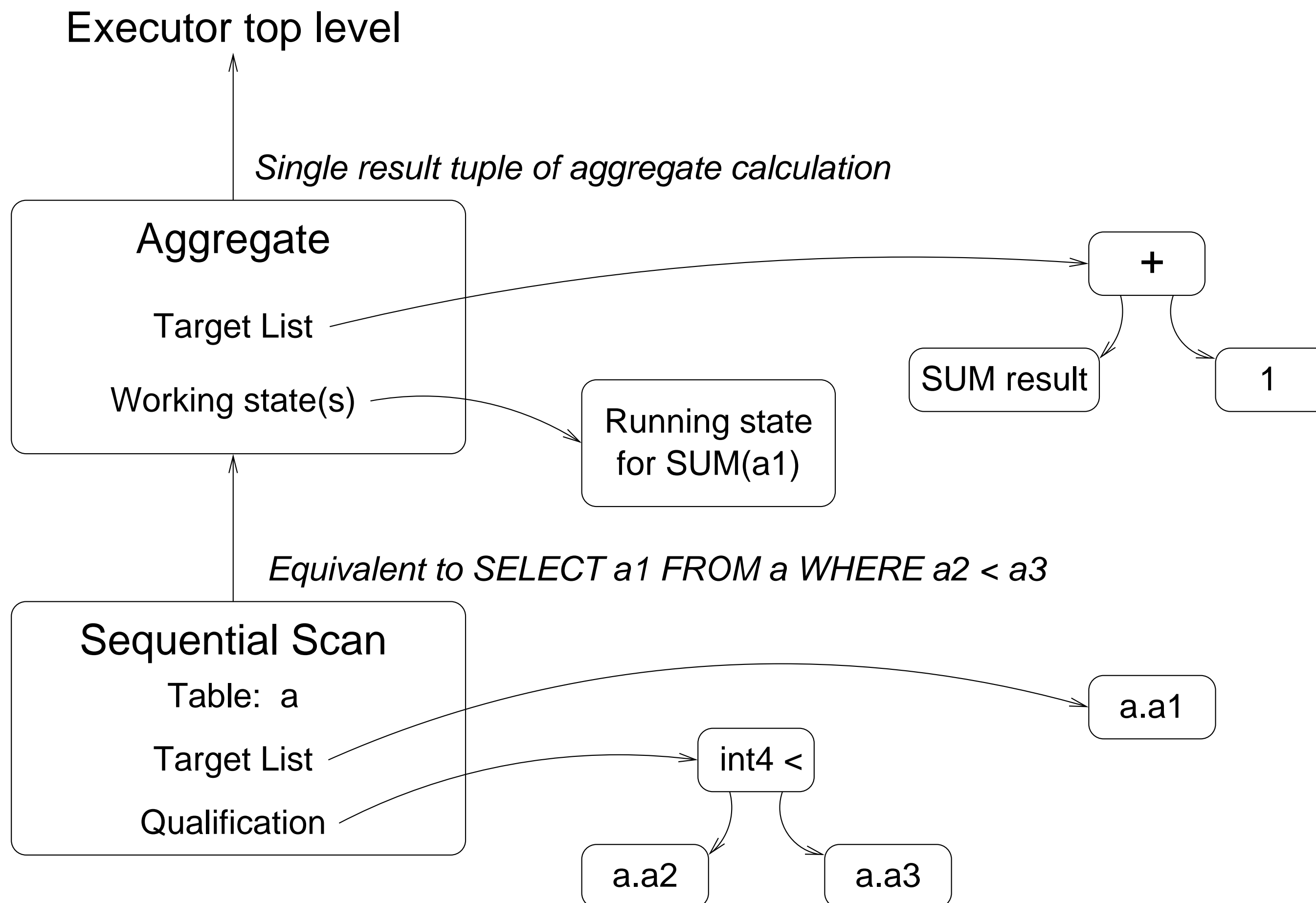
There are also special-purpose node types, such as SORT and AGGREGATE.

Steps of query processing: executor example

Query:

```
SELECT SUM(a1)+1 FROM a WHERE a2 < a3
```

Plan tree:

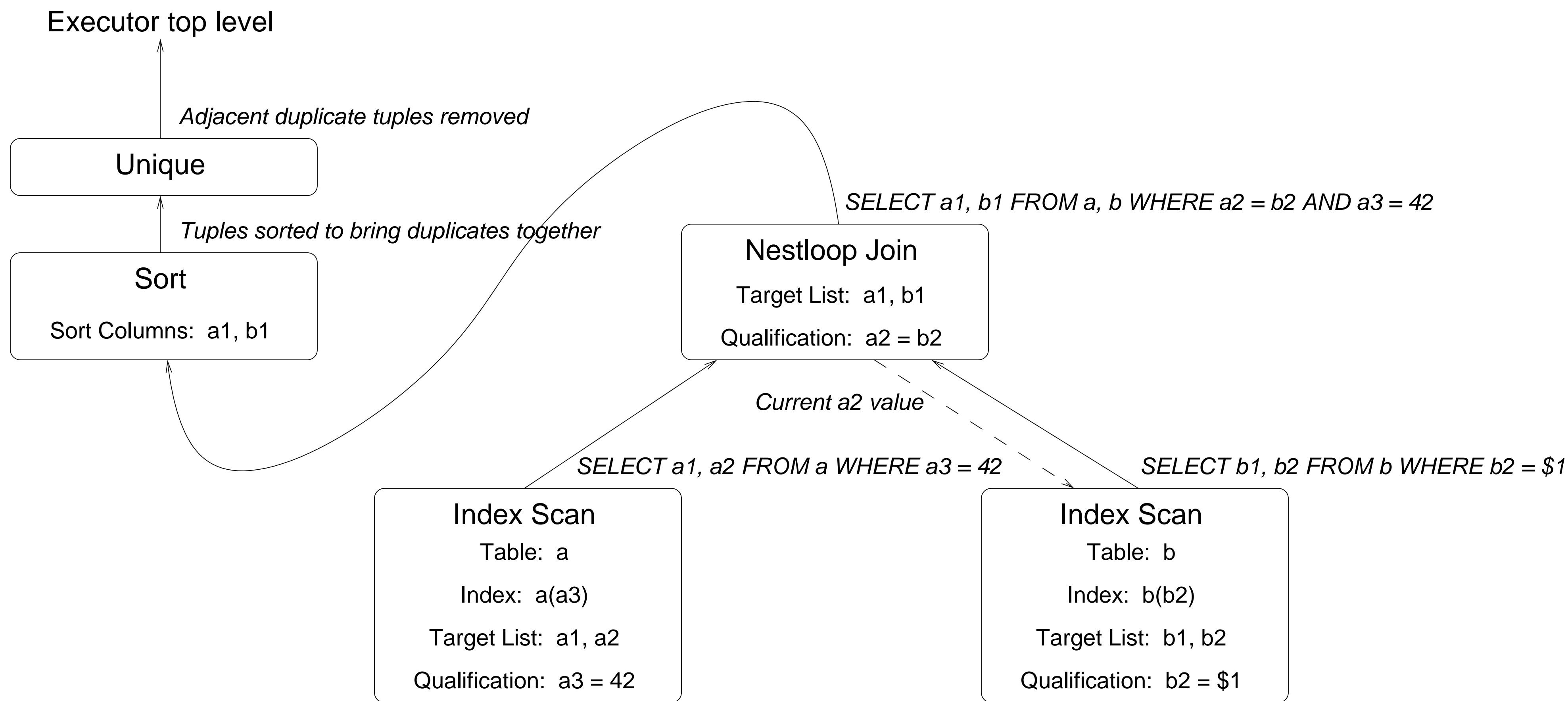


Steps of query processing: executor example

Query:

```
SELECT DISTINCT a1, b1 FROM a, b WHERE a2 = b2 AND a3 = 42
```

Plan tree:



Steps of query processing: execution of non-SELECT queries

What if the query's not a SELECT?

Guess what, the plan tree machinery doesn't really care.

- **INSERT is same as SELECT except for where the result rows go.**

- **For UPDATE/DELETE, the planner adds a hidden targetlist item to return the TID of the selected rows, which the executor's top level uses to determine which row to update or delete.**

So, for planning and most of the executor, all queries look like SELECTs.

Only the executor top level has to behave differently depending on the query type.

Steps of query processing: planner

Now that we've looked at plan trees, we can talk about the planner.

The basic idea of the planner is cost-estimate-based selection of the best plan tree for a given query.

Simple example:

```
SELECT * FROM t WHERE f1 < 100
```

Assuming there is an index on t(f1), two possible execution plans are considered:

- **sequential scan over all of t**
- **index scan with index restriction f1 < 100**

Costs of each plan (in disk page fetches and CPU time) are estimated and the plan with lower estimated cost is selected.

Note that the indexscan is *not* an automatic winner, and should not be. The choice will depend on what fraction of the rows of t are estimated to be retrieved.

Steps of query processing: join planning

In a multi-table query, we first estimate costs for sequential scans and index scans (where applicable) of each component table, then build up a join tree in which all possible pairwise join paths are considered. The *k*'th level of the tree gives us the cheapest ways to join any *k* of the base tables, and at the top level (level *n* for an *n*-table query) we find the cheapest overall join path.

If there are a large number of tables involved (more than about ten), this exhaustive search of the join space is impractical due to exponential growth of the number of possible join paths. In such cases we fall back on a probabilistic search through a limited number of alternatives, using a genetic optimization algorithm.

Steps of query processing: summary

Pros:

- **System usually finds a good query plan without human assistance**

Cons:

- **Plan is only as good as the system's cost models and statistics about user data**

  It's possible to choose a spectacularly bad plan if the statistics are way off base.

  We are working on improving the models and statistics.

- **Time to generate a plan can be annoying, particularly for repetitive queries**

  Pushing complex queries into plpgsql functions helps, since plpgsql caches plans.

  We are also talking about creating a more generic plan cache mechanism.

Summary

PostgreSQL is a complex system that takes a good deal of study to become familiar with.

If you do study it, you find considerable elegance of design.

... much of the credit for which is due to Stonebraker and his students at Berkeley, not to the current developers.

While PostgreSQL has some designed-in limitations, it is also capable of doing things that no competing DBMS can.