Tutorial 1: Internals of PostgreSQL and WiredTiger
Winter 2019
Outline

1. Introduction to PostgreSQL
2. PostgreSQL Architecture
3. PostgreSQL Components
   - Parser
   - Query Rewriter
   - Optimizer
   - Executor
   - Storage
4. WiredTiger
Introduction to PostgreSQL

- PostgreSQL is an open-source, object-relational database system.
- PostgreSQL was first developed at University of California, Berkeley under the name POSTGRES.
- Throughout this course, we will use version 9.3.1 as a code-base to implement new features on top of it.
PostgreSQL Architecture

Client Process (psql / pglib) → Postmaster → Postgres → PostgreSQL backend
Client Types

psql
Psq1 is an interactive client that allows the user to submit SQL queries.

libpq
libpq is the C application programmer's interface (API) to PostgreSQL. libpq is a set of library functions that allow client programs to pass queries to the PostgreSQL backend server.

Server Programming Interface (SPI)
SPI gives writers of user-defined C functions the ability to run SQL commands inside their functions. SPI is a set of interface functions to simplify access to the parser, planner, optimizer, and executor. SPI also does some memory management.

JDBC, ...
PostgreSQL Backend

Reference: Tom Lane, A Tour of PostgreSQL Internals
Query Parser

- The SQL query is tokenized and parsed according to SQL language standard.

- It parses and analyzes the string input and produces a Query structure (Query Tree) for the executor.

- Syntax-errors are caught at this stage.

- Source code located in the directory `src/backend/parser`
Example

Input:

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

Output:

![SELECT Query Diagram]

Reference: Tom Lane, A Tour of PostgreSQL Internals
Query Rewriter

- Also called the *Rule System*.
- It modifies the *Query* structure based on a set of rules before passing it to the optimizer.
- Rules can be user-defined or automatically created for views.
- Rules types are **ON SELECT**, **ON UPDATE/INSERT/DELETE**

- Example:

  
  ```sql
  SELECT * FROM Table1, View2, ...
  ```

  is flattened to

  ```sql
  SELECT * FROM Table1, (SELECT * FROM Table2,Table3...) AS View2 ...
  ```
Query Rewriter - Example

- Trace changes to the `sl_avail` column in the `shoelace_data` relation in a log table:

  ```sql
  CREATE RULE log_shoelace
  AS ON UPDATE TO shoelace_data
  WHERE NEW.sl_avail <> OLD.sl_avail
  DO
  INSERT INTO shoelace_log
  VALUES ( NEW.sl_name,
           NEW.sl_avail,
           current_user,
           current_timestamp );
  ```
PostgreSQL Optimizer

- PostgreSQL uses bottom-up optimization (dynamic programming).

- Optimizer accepts a Query structure and produces a plan with the least estimated cost.

- The planner creates a PlannerGlobal structure to keep track of all the global information for planning and optimization.

- PlannerGlobal is visible to all nodes being optimized.
Optimizing each Relation

- The optimizer builds a RelOptInfo structure for each relation (base or join relation).
- The RelOptInfo structure holds per-relation information for planning and optimization, including:
  - The estimated number of rows produced by the relation
  - A list of Paths, one for each potential method of generating the relation
Path Structure

- Possible **physical plans** to answer the query are stored in a structure named *Path*.
- A Path is a hierarchical structure. Each node represents a query operator.
- A Path specifies the access methods, the join order and the join algorithms used at each node.
**Path Structure**

- Possible **physical plans** to answer the query are stored in a structure named **Path**.
- A Path is a hierarchical structure. Each node represents a query operator.
- A Path specifies the access methods, the join order and the join algorithms used at each node.
- Example path:
Query Operators in Paths

- **Unary** operators: accepts one input relation
  - Sequential Scan/ Index Scan
  - Sort
  - Unique
  - Aggregate
  - Materialize

- **Binary** operators: accepts two input relations
  - Nested Loop Join
  - Hash Join
  - Merge Join
Constructing Paths

● Paths are constructed in a bottom-up style.

● Two main types of relations:
  ○ Base Rel could be primitive tables, or subqueries that are planned via a separate recursive invocation of the planner
  ○ Join Rel is a combination of base rels

● Joinrels are constructed incrementally.
Larger joinrels are constructed by combining smaller baserels and joinrels.
Constructing Paths

- Paths are constructed in a bottom-up style.
- Two main types of relations:
  - **Base Rel** could be primitive tables, or subqueries that are planned via a separate recursive invocation of the planner
  - **Join Rel** is a combination of base rels
- **Joinrels** are constructed incrementally. Larger **joinrels** are constructed by combining smaller baserels and joinrels.
- Example: constructing \( A \bowtie B \bowtie C \bowtie D \)
Path Structure Information

- The optimizer adds all the potentially useful access paths of a relation to a list that is stored in its `RelOptInfo` structure.

- Each `Path` structure contains information about its estimated cost and sort ordering of its output (if any).
Entry Points and Important Files

- Optimizer component is included in the directory
  src/backend/optimizer

- The optimizer entry point is in the file
  src/backend/optimizer/plan/planner.c

- Path construction and cost estimation is included in the directory
  src/backend/optimizer/path

- A README file is included in the optimizer directory for more details.

- You can use the EXPLAIN command to print the selected plan, along with
  estimated and actual statistics (e.g. cardinality, execution time)
Plan Executor

- After finding the path with the least cost, a Plan is constructed from the found path.
- One-to-one mapping between a Path and a Plan
- Plan structure suits the query processing stage
- A Plan is a hierarchical structure of query operators.
Query Operators

- Operators can be classified to two categories:
  - Blocking Operators: sort, aggregate
  - Non-blocking operators (pipelined): Index Scan, Merge join, Nested-Loop Join

- Pipelining allows fast reporting of results; user does not have to wait till all of the input tuples are processed before start getting results.
Tuples Retrieval

- The Executor is based on demand-pull interface.
- There are mainly two types of Plan node execution:

1. single-tuple retrieval
   - A tuple is requested through the function `ExecProcNode`
   - Examples: scan, join, sort

2. multi-tuple retrieval
   - A structure containing all tuples is returned to the caller thought the function `MultiExecProcNode`
   - Examples: bitmap scan, hash
Example
Node Execution State

- ExecProcNode is *reentrant* procedure
- The state of the previous execution (e.g. which tuples are already retrieved) must be stored
- Each Plan node has a corresponding PlanState structure (e.g., HashjoinState)
Storage and WiredTiger
Storage

- The data is stored as files on disk.
- Each relation (table) has its own file.
- Storage engine code is in `backend/storage/smgr`
- File I/O unit is “Block” which is 4kB in size.
Storage

- Layout of blocks within a file for a relation.
- A linked list style structure.
- `md_fd` points to the head of the list.
- Check `MdfdVec` struct in `md.c`
WiredTiger

- Persistent key-value store.
- Good for fast storage device – Flash device, or in-memory.

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key1</td>
<td>value1</td>
</tr>
<tr>
<td>key2</td>
<td>value2</td>
</tr>
<tr>
<td>key3</td>
<td>value3</td>
</tr>
</tbody>
</table>

- Key and values are arbitrary byte streams.
- Works well on many core systems, scales linearly with number of CPUs.
- Suitable for any application that need low latency database access, e.g.:
  - graph-search query scanning dataset in real-time.
  - cache Hadoop data for real-time query on it.
WiredTiger - Fundamentals

• Keys-Value storage Interface: set_key(), get_key(), set_value(), get_value()

• Three basic constructs are: memtable, sstfile, and logfile.
  ○ memtable is an in-memory data structure, new writes are inserted into it and optionally written to the logfile.
  ○ logfile is a sequentially written file on disk.
  ○ when memtable fills up, it is flushed to sstfile on disk and the logfile is deleted.
WiredTiger - How to use it

- Wiredtiger is a C++ library with C/C++ API.
- Basic operations:
  - Create wiredtiger connection: `wiredtiger_open`
  - Open session: `open_session`
  - Create table: `create`
  - Open existing table: `open_cursor`
  - set/get key: `set_key` / `get_key`
  - set/get value: `set_value` / `get_value`
- More about usage at:
  - [http://source.wiredtiger.com/2.9.0/examples.html](http://source.wiredtiger.com/2.9.0/examples.html)
  - [http://source.wiredtiger.com/2.9.0/basic_api.html](http://source.wiredtiger.com/2.9.0/basic_api.html)
PostgreSQL Code Debugging Session