34.5. Pipeline Mode

libpq pipeline mode allows applications to send a query without having to read the result of the previously sent query. Taking advantage of the pipeline mode, a client will wait less for the server, since multiple queries/results can be sent/received in a single network transaction.

While pipeline mode provides a significant performance boost, writing clients using the pipeline mode is more complex because it involves managing a queue of pending queries and finding which result corresponds to which query in the queue.

Pipeline mode also generally consumes more memory on both the client and server, though careful and aggressive management of the send/receive queue can mitigate this. This applies whether or not the connection is in blocking or non-blocking mode.

While the pipeline API was introduced in PostgreSQL 14, it is a client-side feature which doesn't require special server support and works on any server that supports the v3 extended query protocol.

34.5.1. Using Pipeline Mode

To issue pipelines, the application must switch the connection into pipeline mode, which is done with `PQenterPipelineMode`. `PQpipelineStatus` can be used to test whether pipeline mode is active. In pipeline mode, only asynchronous operations that utilize the extended query protocol are permitted, command strings containing multiple SQL commands are disallowed, and so is `COPY`. Using synchronous command execution functions such as `PQfn`, `PQexec`, `PQexecParams`, `PQprepare`, `PQexecPrepared`, `PQdescribePrepared`, `PQdescribePortal`, is an error condition. `PQsendQuery` is also disallowed, because it uses the simple query protocol. Once all dispatched commands have had their results processed, and the end pipeline result has been consumed, the application may return to non-pipelined mode with `PQexitPipelineMode`.

> **Note**
>
> It is best to use pipeline mode with libpq in non-blocking mode. If used in blocking mode it is possible for a client/server deadlock to occur. [15]

34.5.1.1. Issuing Queries

After entering pipeline mode, the application dispatches requests using `PQsendQueryParams` or its prepared-query sibling `PQsendQueryPrepared`. These requests are queued on the client-side until flushed to the server; this occurs when `PQpipelineSync` is used to establish a synchronization point in the pipeline, or when `PQflush` is called. The functions `PQsendPrepare`, `PQsendDescribePrepared`, and `PQsendDescribePortal` also work in pipeline mode. Result processing is described below.

The server executes statements, and returns results, in the order the client sends them. The server will begin executing the commands in the pipeline immediately, not waiting for the end of the pipeline. Note that results are buffered on the server side; the server flushes that buffer when a synchronization point is established with `PQpipelineSync`, or when `PQsendFlushRequest` is called. If any statement encounters an error, the server aborts the current transaction and does not execute any subsequent command in the queue until the next synchronization point; a `PGRES_PIPELINE_ABORTED` result is produced for each such command. (This remains true even if the commands in the pipeline would rollback the transaction.) Query processing resumes after the synchronization point.

It's fine for one operation to depend on the results of a prior one; for example, one query may define a table that the next query in the same pipeline uses. Similarly, an application may create a named prepared statement and execute it with later statements in the same pipeline.

34.5.1.2. Processing Results

To process the result of one query in a pipeline, the application calls `PQgetResult` repeatedly and handles each result until `PQgetResult` returns null. The result from the next query in the pipeline may then be retrieved using `PQgetResult` again and the cycle repeated. The application handles individual statement results as normal. When the results of all the queries in the pipeline have been returned, `PQgetResult` returns a result containing the status value `PGRES_PIPELINE_SYNC`.

The client may choose to defer result processing until the complete pipeline has been sent, or interleave that with sending further queries in the pipeline; see Section 34.5.1.4.

To enter single-row mode, call `PQsetSingleRowMode` before retrieving results with `PQgetResult`. This mode selection is effective only for the query currently being processed. For more information on the use of `PQsetSingleRowMode`, refer to Section 34.6.

`PQgetResult` behaves the same as for normal asynchronous processing except that it may contain the new `PGresult` types `PGRES_PIPELINE_SYNC` and `PGRES_PIPELINE_ABORTED`. `PGRES_PIPELINE_SYNC` is reported exactly once for each `PQpipelineSync` at the corresponding point in the pipeline. `PGRES_PIPELINE_ABORTED` is emitted in place of a normal query result for the first error and all subsequent results until the next `PGRES_PIPELINE_SYNC`; see Section 34.5.1.3.

`PQisBusy`, `PQconsumeInput`, etc operate as normal when processing pipeline results. In particular, a call to `PQisBusy` in the middle of a pipeline returns 0 if the results for all the queries issued so far have been consumed.

libpq does not provide any information to the application about the query currently being processed (except that `PQgetResult` returns null to indicate that we start returning the results of next query). The application must keep track of the order in which it sent queries, to associate them with their corresponding results. Applications will typically use a state machine or a FIFO queue for this.

34.5.1.3. Error Handling

From the client's perspective, after `PQresultStatus` returns `PGRES_FATAL_ERROR`, the pipeline is flagged as aborted. `PQresultStatus` will report a `PGRES_PIPELINE_ABORTED` result for each remaining queued operation in an aborted pipeline. The result for `PQpipelineSync` is reported as `PGRES_PIPELINE_SYNC` to signal the end of the aborted pipeline and resumption of normal result processing.

The client *must* process results with `PQgetResult` during error recovery.

If the pipeline used an implicit transaction, then operations that have already executed are rolled back and operations that were queued to follow the failed operation are skipped entirely. The same behavior holds if the pipeline starts and commits a single explicit transaction (i.e. the first statement is `BEGIN` and the last is `COMMIT`) except that the session remains in an aborted transaction state at the end of the pipeline. If a pipeline contains *multiple explicit transactions*, all transactions that committed prior to the error remain committed, the currently in-progress transaction is aborted, and all subsequent operations are skipped completely, including subsequent transactions. If a pipeline synchronization point occurs with an explicit transaction block in aborted state, the next pipeline will become aborted immediately unless the next command puts the transaction in normal mode with `ROLLBACK`.

> **Note**
>
> The client must not assume that work is committed when it *sends* a `COMMIT` — only when the corresponding result is received to confirm the commit is complete. Because errors arrive asynchronously, the application needs to be able to restart from the last *received* committed change and resend work done after that point if something goes wrong.

34.5.1.4. Interleaving Result Processing And Query Dispatch

To avoid deadlocks on large pipelines the client should be structured around a non-blocking event loop using operating system facilities such as `select`, `poll`, `WaitForMultipleObjectEx`, etc.

The client application should generally maintain a queue of work remaining to be dispatched and a queue of work that has been dispatched but not yet had its results processed. When the socket is writable it should dispatch more work. When the socket is readable it should read results and process them, matching them up to the next entry in its corresponding results queue. Based on available memory, results from the socket should be read frequently: there's no need to wait until the pipeline end to read the results. Pipelines should be scoped to logical units of work, usually (but not necessarily) one transaction per pipeline. There's no need to exit pipeline mode and re-enter it between pipelines, or to wait for one pipeline to finish before sending the next.

An example using `select()` and a simple state machine to track sent and received work is in `src/test/modules/libpq_pipeline/libpq_pipeline.c` in the PostgreSQL source distribution.

34.5.2. Functions Associated with Pipeline Mode

`PQpipelineStatus`

Returns the current pipeline mode status of the libpq connection.

```
PGpipelineStatus PQpipelineStatus(const PGconn *conn);
```

`PQpipelineStatus` can return one of the following values:

`PQ_PIPELINE_ON`

The libpq connection is in pipeline mode.

`PQ_PIPELINE_OFF`

The libpq connection is *not* in pipeline mode.

`PQ_PIPELINE_ABORTED`

The libpq connection is in pipeline mode and an error occurred while processing the current pipeline. The aborted flag is cleared when `PQgetResult` returns a result of type `PGRES_PIPELINE_SYNC`.

`PQenterPipelineMode`

Causes a connection to enter pipeline mode if it is currently idle or already in pipeline mode.

```
int PQenterPipelineMode(PGconn *conn);
```

Returns 1 for success. Returns 0 and has no effect if the connection is not currently idle, i.e., it has a result ready, or it is waiting for more input from the server, etc. This function does not actually send anything to the server, it just changes the libpq connection state.

`PQexitPipelineMode`

Causes a connection to exit pipeline mode if it is currently in pipeline mode with an empty queue and no pending results.

```
int PQexitPipelineMode(PGconn *conn);
```

Returns 1 for success. Returns 1 and takes no action if not in pipeline mode. If the current statement isn't finished processing, or `PQgetResult` has not been called to collect results from all previously sent query, returns 0 (in which case, use `PQerrorMessage` to get more information about the failure).

`PQpipelineSync`

Marks a synchronization point in a pipeline by sending a sync message and flushing the send buffer. This serves as the delimiter of an implicit transaction and an error recovery point; see Section 34.5.1.3.

```
int PQpipelineSync(PGconn *conn);
```

Returns 1 for success. Returns 0 if the connection is not in pipeline mode or sending a sync message failed.

`PQsendFlushRequest`

Sends a request for the server to flush its output buffer.

```
int PQsendFlushRequest(PGconn *conn);
```

Returns 1 for success. Returns 0 on any failure.

The server flushes its output buffer automatically as a result of `PQpipelineSync` being called, or on any request when not in pipeline mode; this function is useful to cause the server to flush its output buffer in pipeline mode without establishing a synchronization point. Note that the request is not itself flushed to the server automatically; use `PQflush` if necessary.

34.5.3. When to Use Pipeline Mode

Much like asynchronous query mode, there is no meaningful performance overhead when using pipeline mode. It increases client application complexity, and extra caution is required to prevent client/server deadlocks, but pipeline mode can offer considerable performance improvements, in exchange for increased memory usage from leaving state around longer.

Pipeline mode is most useful when the server is distant, i.e., network latency ("ping time") is high, and also when many small operations are being performed in rapid succession. There is usually less benefit in using pipelined commands when each query takes many multiples of the client/server round-trip time to execute. A 100-statement operation run on a server 300 ms round-trip-time away would take 30 seconds in network latency alone without pipelining; with pipelining it may spend as little as 0.3 s waiting for results from the server.

Use pipelined commands when your application does lots of small `INSERT`, `UPDATE` and `DELETE` operations that can't easily be transformed into operations on sets, or into a `COPY` operation.

Pipeline mode is not useful when information from one operation is required by the client to produce the next operation. In such cases, the client would have to introduce a synchronization point and wait for a full client/server round-trip to get the results it needs. However, it's often possible to adjust the client design to exchange the required information server-side. Read-modify-write cycles are especially good candidates; for example:

```
BEGIN;
SELECT x FROM mytable WHERE id = 42 FOR UPDATE;
-- result: x=2
-- client adds 1 to x:
UPDATE mytable SET x = 3 WHERE id = 42;
COMMIT;
```

could be much more efficiently done with:

```
UPDATE mytable SET x = x + 1 WHERE id = 42;
```

Pipelining is less useful, and more complex, when a single pipeline contains multiple transactions (see Section 34.5.1.3).

---

[15] The client will block trying to send queries to the server, but the server will block trying to send results to the client from queries it has already processed. This only occurs when the client sends enough queries to fill both its output buffer and the server's receive buffer before it switches to processing input from the server, but it's hard to predict exactly when that will happen.

Submit correction

If you see anything in the documentation that is not correct, does not match your experience with the particular feature or requires further clarification, please use this form to report a documentation issue.

Privacy Policy | Code of Conduct | About PostgreSQL | Contact
Copyright © 1996-2022 The PostgreSQL Global Development Group