JDBC Performance from the Inside

Mar, 2017

Introduction

- Dave Cramer
- Work for Pivotal on the Greenplum database project https://github.com/greenplum-db/gpdb
- Maintainer for the JDBC driver since 1999
- Overheard conversation about how poorly the driver performed because of how it was built.
- Allegedly was very CPU intensive because of all the inheritance
- Things have changed a lot since then so I thought this would be a good idea for a talk, gather some statistics and show all the performance gains as a result of simplifying the code

Pivotal

Overview

- History of the driver
- Previous source layout
- Typical usage pattern
- Using Prepared Statements
- Batch processing how and why
- Optimal Fetch Size
- Latest Release

Pivotal

History

- Originally written by Peter Mount in 1997
- Supported JDBC 1.2
- 1997 JDBC 1.2 Java 1.1
- 1999 JDBC 2.1 Java 1.2
- 2001 JDBC 3.0 Java 1.4
- 2006 JDBC 4.0 Java 6
- 2011 JDBC 4.1 Java 7
- 2014 JDBC 4.2 Java 8
- 2017 JDBC 4.3 Java 9 (Maybe ?)
- Each one of these were incremental additions to the interface
- Requiring additional concrete implementations of the spec to be implemented

Pivotal

Source code layout

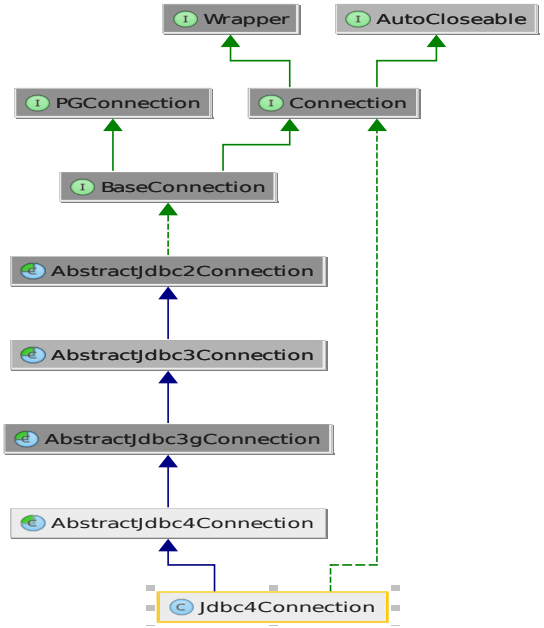
Pivotal

Before Maven (the ant years)

- jdbc2->jdbc3->jdbc3g->jdbc4->jdbc42
- Each one of these had abstract implementations, and concrete implementations
- Which one was built was determined by filters using ant
- Lions share of code was in jdbc2 package
- This meant that a concrete jdbc42 implementation public class Jdbc42Connection extended AbstractJdbc42Connection which extends **AbstractJdbc4Connection** which extends **AbstractJdbc3gConnection** which extends **AbstractJdbc3Connection** which extended **AbstractJdbc2Connection**

Pivotal

Before Maven (the ant years)



Before Maven (the ant years)

- Why didn’t you just use –target and compile previous versions with the latest compiler
- In theory since older versions of the spec will never attempt to access more recent functions in an interface this should “just work”
- Well embarrassingly we didn’t think of it.
- Up until Java 8 this was possible.. The JDBC spec was never supposed to introduce a backward incompatibility.
- In Java 8 they added java.time.* the problem is: attempting to load a driver using an earlier JDK with java.time in it will cause a ClassNotFound Exception. We are required to be able to pass a java.time.* object into setObject

Pivotal

Mavenizing the driver

- Why ? I was pretty hesitant to essentially rewrite the driver
- Easier to just include in your project

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.0.0</version> <!-- Java 8 />
</dependency>
```

- Still have the same problems they are just solved differently
- Ant had filters to filter out which files are compiled for each build
- Maven uses pre-processing to add or remove code, avoids the multiple class extension
- *//#if mvn.project.property.postgresql.jdbc.spec >= "JDBC4.2"*

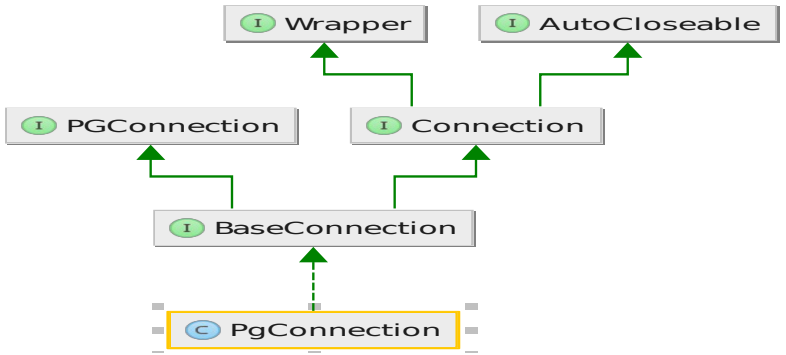
Pivotal

Mavenizing the driver

- Code re-organized release 1207 Dec 2015
- All of the abstract class machinations have been removed
- One class file works for all versions of JDBC
- Reduces CPU load
- Real advantage to mavenizing the project. The code is much simpler.
- Easier to debug, can be easily loaded into an IDE
- More people have provided Pull Requests

Pivotal

After Maven



Time to test the hypothesis

- Borrowed some code from https://github.com/8kdata/javapgperf
- Plug for ToroDB https://github.com/torodb/torodb
- **CREATE TABLE IF NOT EXISTS** number **AS**
  SELECT i, **'Hello there '** || i **AS** t, **'{"i": '** || i || **', "t": "'** || **'Hello there '** || i || **'" }'**
  AS j
  FROM *generate_series*(1,10 * 1000 * 1000) **AS** i;
- Column | Type | Modifiers
  --------+---------+-----------
  i | integer |
  t | text |
  j | text |

What do the tests do ?

- 1_int select i from number
- 2_String select t from number
- 3_IntString select i,t from number
- 4_IntStringJson select i, t, j from number
- 5_IntStringColumnNumber select i,t and use column number instead of column name
- 6_StringNoAutocommit select t from number
- Ran the test suite for a number of different JDBC versions 1204, 1208, 1210, 1212, 42.0.0

What do the tests do ?

```
public class _4_IntStringJson {
    public static final String QUERY = "SELECT i, t, j FROM number";

    private class JsonElements {
        private int i;
        private String t;
    }

    @Benchmark
    public void test(Blackhole blackhole, PgStatStatements pgStatStatements) throws SQLException {
        pgStatStatements.setTestName(QueryBenchmarks.JMHTestNameFromClass(_4_IntStringJson.class));

        Gson gson = new Gson();

        QueryUtil.executeProcessQuery(QUERY, resultSet -> {
                while (resultSet.next()) {
                    blackhole.consume(resultSet.getInt("i"));
                    blackhole.consume(resultSet.getString("t"));
                    blackhole.consume(gson.fromJson(resultSet.getString("j"), JsonElements.class));
                }
        });
    }
}
```

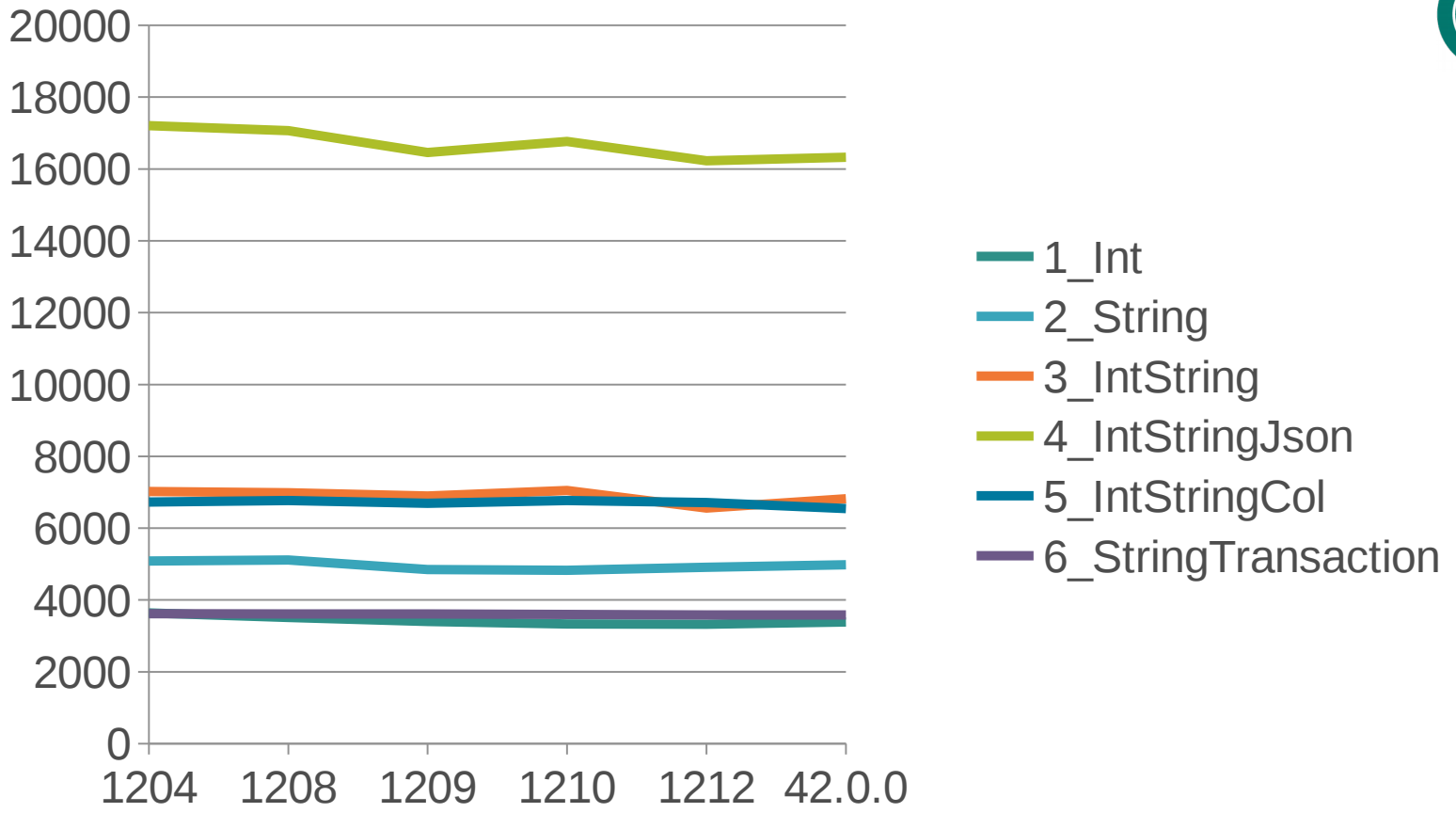
```
public class Main {
    private static final int ITERATIONS = 20;
    // Help profiling with sampling agents
    private static final int NO_FORKS_RUN_ON_THE_SAME_JVM = 0;

    public static void main(String[] args) throws RunnerException {
        if(args.length != 1 || args[0] == null || args[0].isEmpty()) {
            System.exit(1);
        }
        String testName = args[0];

        Options opt = addTestToOptionsBuilder(new OptionsBuilder(), testName)
            .addProfiler(org.postgresql.benchmark.profilers.FlightRecorderProfiler.class)
            //.forks(1)
            //.jvmArgsPrepend("-Xmx128m")
            // We need to avoid warmup iterations as they however counts towards total Postgres time
            .warmupIterations(0)
            .measurementIterations(ITERATIONS)
            .timeUnit(TimeUnit.MILLISECONDS)
            .mode(Mode.SingleShotTime)
            .verbosity(VerboseMode.SILENT)
            .build();

        Collection<RunResult> runResults = new Runner(opt).run();

        runResults.stream().forEach(runResult ->
            System.out.printf(
                    "Java:\t%s\t%.2f\n",
                    runResult.getParams().getBenchmark(),
                    runResult.getPrimaryResult().getScore()
            )
        );
    }
```



Something good came out of Maven

- Well it didn’t really improve performance but:
- It is Easier to understand
- More people are working on it
- Easier to work with simply import the maven pom.xml into Intellij
- Easier to push to maven

Some things that really did improve performance

- Set Fetch Size
- Fixed deadlocks
- Insert rewrite
- Fixing bugs

Set FetchSize performance

- Fetch a large amount of data with different fetch sizes

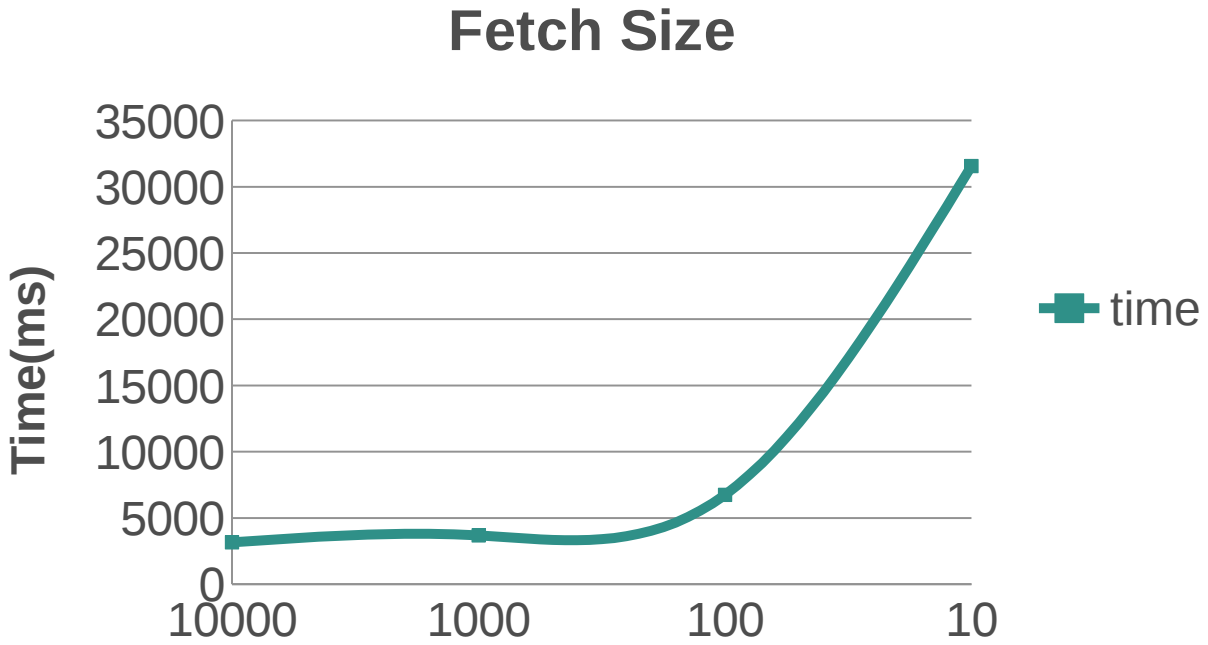
```
public static final String QUERY = "SELECT t FROM number";

@Benchmark
public void test(Blackhole blackhole, PgStatStatements pgStatStatements) throws SQLException {
    pgStatStatements.setTestName(QueryBenchmarks.JMHTestNameFromClass(_6_String_NoAutocommit.class));

    QueryUtil.executeProcessQueryNoAutocommit(QUERY, resultSet -> {
            while (resultSet.next()) {
                blackhole.consume(resultSet.getString(1));
            }
    });
}

// Used to fetch rows in batches from the db. Will only work if the connection does not use AutoCommit
PGProperty.DEFAULT_ROW_FETCH_SIZE.set(properties, FETCH_SIZE);
```

setFetchSize



Inserting batch Deadlock

- Ideally we would like to:
- Insert N rows of inserts where N is some arbitrarily large number
- PARSE S_1
- BIND/EXEC N TIMES
- DEALLOCATE

Unfortunately this doesn’t work

- Driver is busy sending data, so it hasn’t retrieved any responses
- BIND/EXEC only sends data it does not read it
- Server is busy sending responses, so it can’t fetch any more insert queries
- So we have a situation where the driver is continually sending, and the server is continually sending. Neither one is reading the responses.

Pivotal

Every so often we have to sync

- Parse S_1
- BIND/EXEC
- BIND/EXEC
- SYNC … flush and wait for response
- The more sync's the slower it performs
- Current code sync's every 64k of data

What are the options for inserting lots of data

- For each row insertExecute
- For each row insertBatch
- Insert values (row1), (row2), … (rowN) hand rolled code
- copy

JDBC micro benchmark suite

- Java 1.8_60
- Core i7 2.8GHz
- PostgreSQL 9.6 (beta1)
- https://github.com/pgjdbc/pgjdbc/tree/master/ubenchmark
- create table batch_perf_test(a int4, b varchar(100), c int4)

```
Table "public.batch_perf_test"
Column |          Type          | Modifiers
--------+------------------------+-----------
a      | integer                |
b      | character varying(100) |
c      | integer                |
```

The Code

- ```
  public int[] insertBatch() throws SQLException {
    if (p2multi > 1) {
      // Multi values(),(),() case
      for (int i = 0; i < p1nrows; ) {
        for (int k = 0, pos = 1; k < p2multi; k++, i++) {
          ps.setInt(pos, i);
          pos++;
          ps.setString(pos, strings[i]);
          pos++;
          ps.setInt(pos, i);
          pos++;
        }
        ps.addBatch();
      }
  ```

- If we have 10 rows and p2multi is 2 the outer loop is executed 5 times and we insert 2 rows at a time

- Insert into foo (a,b,c) values (?,?,?), (?,?,?)
  ```

INSERT Batch where p1multi =1

- For each row Insert into perf (a,b,c) values (?,?,?)
- After N rows executeBatch

INSERT Batch where p1multi >1

- For each row Insert into perf (a,b,c) values (?,?,?), (?,?,?), (?,?,?), (?,?,?)
- After N/p2multi rows executeBatch
- Given 1000 (N) rows if we insert them 100(p2multi) at a time, end up inserting 10 rows 100 wide
- More data inserted per statement, less statements

Pivotal

INSERT Batch with insertRewrite

- For each row Insert into perf (a,b,c) values (?,?,?)
- After N rows executeBatch
- Same as insertBatch except we set the connection parameter insertRewrite=true
- As of 1209 this is has been enabled
- Same as the previous slide except the driver does it for you.

Pivotal

Copy

- Loop over the rows creating the input string in memory
- Build a string in memory which looks like 0\ts0\t0\n1\ts1\t1\n….
- The string will end up being nrows/p2multi long
- Use the copy API to copy this into the table

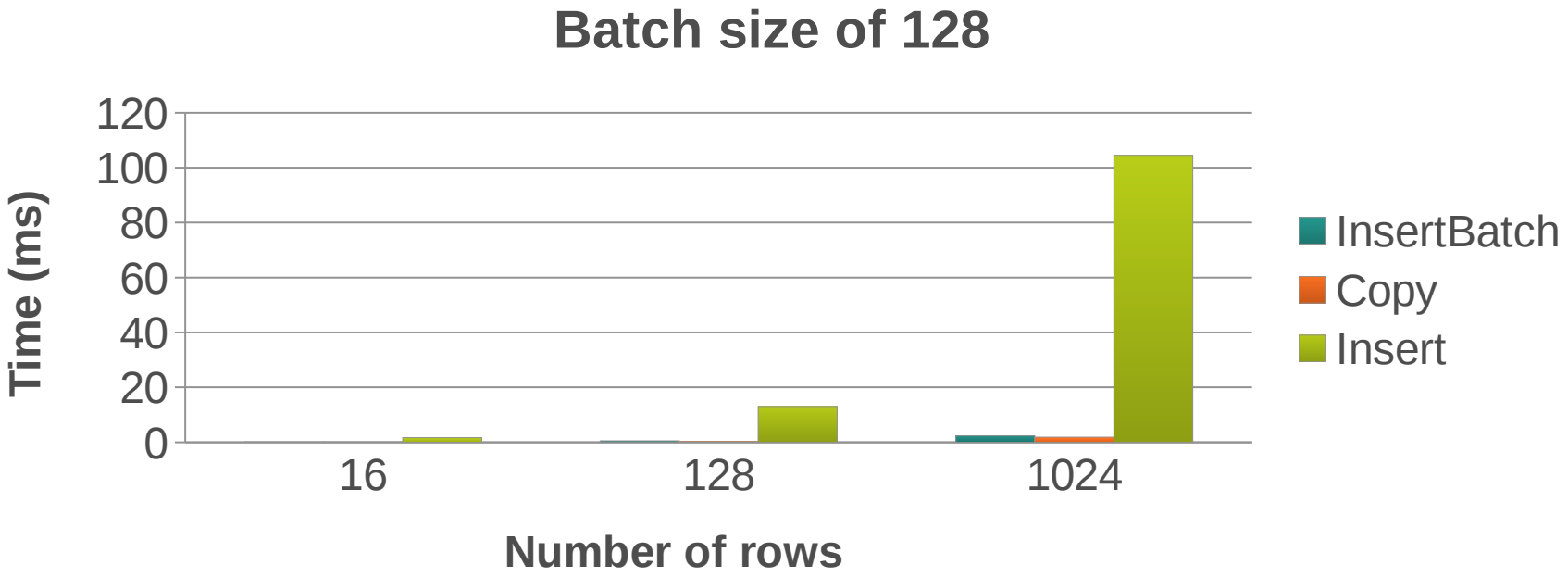
Hand rolled insert struct

- Insert into batch_perf_test select * from unnest (?::batch_perf_test[])
- For N rows setString to ‘{“(1,s1,1)”,”(2,s2,2)”,”(3,s3,3)”}’
- Add Batch
- executeBatch
- The query that gets executes look like:

```
Insert into batch_perf_test
                    select *
                   from unnest (‘{“(1,s1,1)”,”(2,s2,2)”,”(3,s3,3)”}’::batch_perf_test[])
```

Pivotal

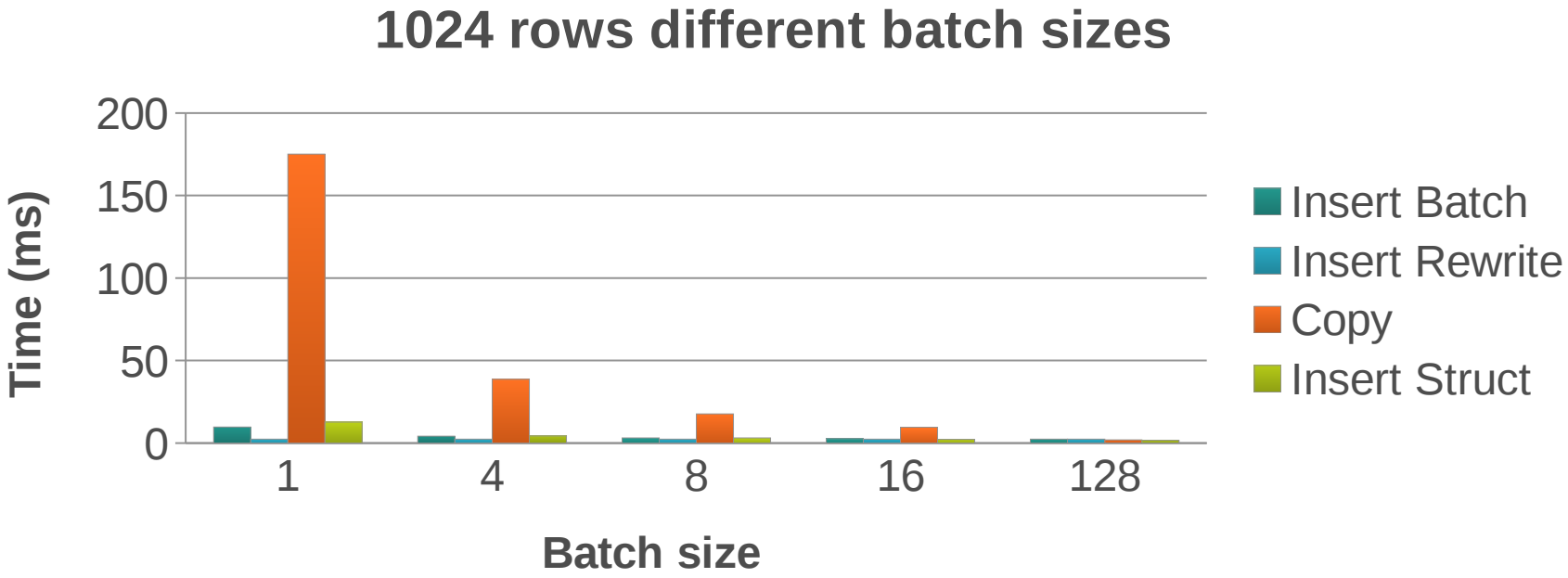
Results



Conclusion

- Compared to batch inserts, plain inserts are very slow for large amounts of data

Results



Bug fixes

- https://github.com/pgjdbc/pgjdbc/pull/380
- QUERY_FORCE_DESCRIBE_PORTAL shared the same value as QUERY_DISABLE_BATCHING effectively disabling batch inserts
- 10x increase in throughput

New Release 42.0.0

- Wanted to divorce ourselves from the server release schedule
- Wanted to reduce confusion as to which version to use. Previously the numbers 9.x were in the version number.
- Introduce semantic versioning
- 42 more or less at random, but also the answer to the question.

Pivotal

Notable changes

- Support dropped for versions before 8.2
- Replace hand written logger with java.util.logging
- Replication protocol API was added.

Logging

- Setting using url, or properties file
- Root logger is org.postgresql
- Properties via URL are loggerLevel and loggerFile
- loggerLevel can be one of: OFF, DEBUG, TRACE
- Corresponds to OFF, FINE, FINEST
- Also possible to use logging.properties file as per normal

Pivotal

Logical Replication

- Create a replication connection
- Create a logical replication slot
- Read changes
- Send confirmation of changes read
- GOTO read more changes

Pivotal

Create a Replication Connection

```
String url = "jdbc:postgresql://localhost:5432/postgres";
 Properties props = new Properties();
 PGProperty.USER.set(props, "postgres");
 PGProperty.PASSWORD.set(props, "postgres");
 PGProperty.ASSUME_MIN_SERVER_VERSION.set(props, "9.4");
 PGProperty.REPLICATION.set(props, "database");
 PGProperty.PREFER_QUERY_MODE.set(props, "simple");
 Connection con = DriverManager.getConnection(url, props);
 PGConnection replConnection = con.unwrap(PGConnection.class);
```

- PGProperty.REPLICATION set to “database” instructs the walsender to connect to the database in the url and allow the connection to be used for logical replication.
- PREFER_QUERY_MODE needs to be set to simple as replication does not allow the use of the extended query mode

Pivotal

Create a Logical Replication Slot

```
try (PreparedStatement preparedStatement =
            connection.prepareStatement("SELECT * FROM pg_create_logical_replication_slot(?, ?)"))
{
    preparedStatement.setString(1, slotName);
    preparedStatement.setString(2, outputPlugin);
    try (ResultSet rs = preparedStatement.executeQuery())
    {
        while (rs.next())
        {
            System.out.println("Slot Name: " + rs.getString(1));
            System.out.println("Xlog Position: " + rs.getString(2));
        }
    }
}
```

- Slots require a name and an output plugin
- Any unique name will work
- The output plugin is a previously compiled C library which formats the logical WAL

Read Changes from database

```
PGReplicationStream stream =
        pgConnection
                .getReplicationAPI()
                .replicationStream()
                .logical()
                .withSlotName(SLOT_NAME)
                .withStartPosition(lsn)
                .withSlotOption("include-xids", true)
                .start();
```

- Open a PGReplicationStream with the same slot name
- Start position can be an existing LSN or InvalidLSN
- SlotOptions are sent to the logical decoder and are decoder specific

Read Changes from database

```
while (true) {
     //non blocking receive message
     ByteBuffer msg = stream.readPending();

     if (msg == null) {
       TimeUnit.MILLISECONDS.sleep(10L);
       continue;
     }

     int offset = msg.arrayOffset();
     byte[] source = msg.array();
     int length = source.length - offset;
     System.out.println(new String(source, offset, length));

     //feedback
     stream.setAppliedLSN(stream.getLastReceiveLSN());
     stream.setFlushedLSN(stream.getLastReceiveLSN());
   }
```

- Read from the stream, data will be in a ByteBuffer
- After reading the data send confirmation messages

How not to use JDBC (unfortunately typical)

- Open connection
- Prepare statement ‘select * from foo where id=?’
- preparedStatment.executeQuery()
- preparedStatement.close()
- Close Connection
- Without a pool connection creation is a heavyweight operation. PostgreSQL uses processes so each connection is a process
- Does not take advantage of caching

Pivotal

Better solution

- Open connection
- Prepare statement ‘select * from foo where id=?’
- By default after 5 executions will create a named statement PARSE S_1 as ‘select * from foo where id=?’
- Multiple preparedStatment.executeQuery() BIND/EXEC instead of PARSE/BIND/EXEC
- Never close the statement if possible

Query cache best practices

- Client side query cache only works in 9.4.1203 and up
- Do not use generated queries, as they generate new server side prepared statement
- Things like executeUpdate('**insert into foo (i,l,f,d) values (1,2,3,4)**') will never use a named statement
- Do not change the type of a parameter as this leads to DEALLOCATE/PREPARE
- Pstmt.setInt(1,1)
- Pstmt.setNull(1,Types.VARCHAR) this will cause the prepared statement to be deallocated

Pivotal

Less obvious issues

- Server Prepare activated after 5 executions
- There is a configuration parameter called prepareThreshold (default 5)
- PGStatement.isUseServerPrepare() can be used to check
- After 5 executions of the same prepared statement we change from unnamed statements to named
- Named statements will use binary mode where possible;
- binary mode is faster when we have to parse things like timestamps
- Named statements are only parsed once on the server then bind/execute operations on the server

setFetchSize

- If we don’t use a fetch size we will read the entire response into memory then process
- Optimizing the data sent at one time reduces memory usage and GC
- Only works with in a transaction
- Make sure fetch size is above 100
- If you have a lot of data this is really the only way to read it in without an Out Of Memory Exception

Pivotal

Performance enhancements review

- Cache parsed statements across PrepareStatement calls now don’t have to parse the statement in java each time
- Execute Batch changed to not execute statement by statement bug in code disabled batching
- Rewrite Batched inserts rewrites inserts from multiple insert into foo (a,b,c) values (1,2,3) to insert into foo (a,b,c) values (1,2,3), (4,5,6) this provides 2x-3x speed up
- Avoid Calendar cloning provides 4x speed increase for setTimestamp pr 376

Conclusions

- Using insert rewrite gives us a 2-3x performance increase for batch inserts
- Makes sense as it is one trip
- Use setFetchSize(100) or greater and use transactions
- Don’t close prepared statements.

https://github.com/pgjdbc/pgjdbc

- Credit where credit is due:
- Much of the optimization work on the driver was done by Vladimir Sitnikov
- Much (if not all) of the work to convert the build to Maven was done by Stephen Nelson
- Rewriting batch statements thanks to Jeremy Whiting
- Replication support was provided by Vladimir Gordiychuk
- Questions ?

Pivotal