JDBC TIPS AND TRICKS

Montreal Meetup Jan 21/2020

Dave Cramer
January 2020

CRUNCHY
Enterprise PostgreSQL

Crunchy Data

Leading provider of trusted open source PostgreSQL technology, support and training.

PostgreSQL
the world's most advanced open source database

Powering Innovation With The World's Most Advanced Open Source Database

Crunchy Enterprise PostgreSQL

Introduction

- Dave Cramer
- Maintainer for the JDBC driver since 1999
- There are many options for connecting
- Many of them I didn’t totally understand
- This talk hopes to unveil some of the more interesting ones
- And explain how they work.

Overview

- Connecting to the driver
- Connection options that change the behaviour
- Performance tips
- Logical Decoding

Connecting to the server

Connection Properties

- PG_DBNAME
- PG_DBHOST
- PG_DBPORT
- USER
- PASSWORD

These can be used in the following manner

```
Properties props = new Properties();
props.setProperty(PGProperty.PG_DBNAME.getName(),"test");
props.setProperty(PGProperty.PG_HOST.getName(),"localhost");
props.setProperty(PGProperty.PG_PORT.getName(),"5432");
props.setProperty(PGProperty.USER.getName(),"davec");
props.setProperty(PGProperty.PASSWORD.getName(), "");
Connection connection = DriverManager.getConnection("jdbc:postgresql:”, props);
```

URL options

- jdbc:postgresql:
  - Connects to localhost, port 5432, database specified in user
- jdbc:postgresql://host/
  - Connects to <host>, port 5432, and database specified in user
- jdbc:postgresql://host:port/
  - Connects to <host><port> and database specified in user
- jdbc:postgresql:database
  - Connects to localhost, port 5432 database “database”
- jdbc:postgresql://host:port/database
  - Specify the host, port and database

Connection fail over

- jdbc:postgresql://host1:port, host2:port/database?targetType=primary|secondary|preferSecondary
- Allows you to specify multiple servers for the driver to connect to
- Provides a way to prefer a server type.

Connection Failover tuning

- targetServerType=primary, secondary, preferSecondary
- Observes if server allows writes to chose
- loadBalanceHosts=boolean will randomly pick from suitable candidates
- hostRecheckSeconds=number of seconds between checking status (read or write) of hosts default is 10 seconds

Logging

- loggerLevel = OFF|DEBUG|TRACE
    - Enables java.util.logging.Logger DEBUG=FINE, TRACE=FINEST
    - Not intended for SQL logging but rather to debug the driver
- loggerFile=<filename> the file to output the log to. If this is not set then the output will be written to the console.

Logging

- FINEST: FE=> SimpleQuery(query="select 1")
- Mar 09, 2018 1:20:33 PM org.postgresql.core.v3.QueryExecutorImpl receiveFields
- FINEST: <=BE RowDescription(1)
- Mar 09, 2018 1:20:33 PM org.postgresql.core.v3.QueryExecutorImpl receiveFields
- FINEST: Field(?column?,INT4,4,T)
- Mar 09, 2018 1:20:33 PM org.postgresql.core.v3.QueryExecutorImpl processResults
- FINEST: <=BE DataRow(len=1)

Logging continued

- We will honour DriverManager.setLogStream or DriverManager.setLogWriter
- Parent logger is org.postgresql
- Since we are using java.util.Logging, we can use a properties file to configure logging

Logging properties file

- handlers=java.util.logging.FileHandler
- org.postgresql.level=FINEST
- java –Djava.util.logging.config.file=…
- handlers=java.util.logging.ConsoleHandler
- org.postgresql.level=ALL
- org.postgresql.Driver.level=INFO
- org.postgresql.core.v3.level=FINE

Find connection leaks

- logUnclosedConnections=boolean
- Provides an easy way to find connection leaks
- If this is turned on we track connection opening by saving a Throwable when the connection is opened. If the finalizer is reached and the connection is still open the stacktrace message created when the connection was opened is printed out.

Autosave

- autosave = never | always | conservative
- PostgreSQL transaction semantics are all or nothing. This is not always desirable
- autosave=always will create a savepoint for every statement in a transaction.
- The effect of which means that if you do
- Insert into invoice_header …
- Insert into invoice_lineitem …
- If the insert into invoice lineitem fails the header will still be valid.
- In conservative mode if the driver determines that reparsing the query will work then it will be reparsed and retried.

Binary Transfer

- binaryTransferEnable=comma separated list of oid’s or names
- binaryTransferDisable
- Currently the driver will use binary mode for most built-in types.

SimpleQueryMode

- Client sends an SQL Command(s)
- Server replies with RowDescription
  - Each column has a name
  - Oid of table
  - Attribute (column) number
  - Type Oid
  - Binary or Text
- Server sends all of the data rows
- Server CommandComplete and ReadyForQuery

ExtendedQuery

- Parse
  - Send query string with placeholders, and parameter types (can be named)
  - Response is ParseOK
- Bind
  - Each parameter has format(binary or text) and value
  - Response is BindOK

ExtendedQuery

- Describe (RowDescription)
  - Response is RowDescriptions
  - Number of fields, field name, type, text or binary
- Execute
  - Responds with DataRows and CommandComplete
- Sync
  - Responds with ReadyForQuery

preferQueryMode

- simple
  - Fewer round trips to db no bind, no parse
  - Required for replication connection
- extended
  - Default creates a server prepared statement, uses parse, bind and execute.
  - Protects against SQL injection
  - Possible to re-use the statement

preferQueryMode

- extendedForPrepared
  - Does not use extended for statements, only prepared statements
  - Potentially faster execution of statements
- extendedCacheEverything
  - Uses extended and caches even simple statements such as ‘select a from tbl’ which is normally not cached

defaultRowFetchSize=int

- Default is 0 which means fetch all rows
  - This is sometimes surprising and can result in out of memory errors
- If set *AND* autocommit=false THEN will limit the number of rows per fetch
- Potentially significant performance boost

stringtype=varchar|unspecified

- Default is varchar, which tells the server that strings are actually strings
- You can use stringtype=‘unspecified’
  - Useful if you have an existing application that uses setString(‘1234’) to set an integer column.
  - Server will attempt to cast the string to the appropriate type.

ApplicationName=String

- sets the application name
- Servers version 9.0 and greater
- Useful for logging and seeing which connections are yours in pg_stat_activity, etc.

readOnly=boolean

- The default is false
- True sends SET SESSION CHARACTERISTICS AS TRANSACTION READ ONLY to the server.
- This blocks any writes to persistent tables, interestingly you can still write to a temporary table.

disableColumnSanitizer=boolean

- columnSanitizer folds column names to lower case.
- Column names like FirstName become firstname.
- Resultset.getInt(“firstname”)
- default is to sanitize names

assumeMinServerVersion=String

- Currently there are only 2 use cases
- 9.0 which will enable
- ApplicationName=ApplicationName (defaults to PostgreSQL JDBC Driver)
- sets extra float digits to 3
- 9.4 necessary for replication connections

currentSchema=someschema

- by default the current schema will be "public"
- If you want to refer to a table in a different schema it would have to be specified by schema.table
- If you set this connection property to "audit" for example instead of "select * from audit.log" you could use select * from log;

reWriteBatchedInserts=true

- Enables the driver to optimize batch inserts by changing multiple insert statements into one insert statement.
- insert into tab1 values (1,2,3);
- insert into tab1 values (4,5,6);
- Rewritten as “insert into tab1 values (1,2,3), ..(4,5,6)

replication=false, true, database

- True tells the backend to go into walsender mode
- Setting to database enables logical replication for that database
- Simple query mode, subset of commands
- Must be accompanied by assumMinServerVersion="9.4" and preferQueryMode="simple"

Performance tricks

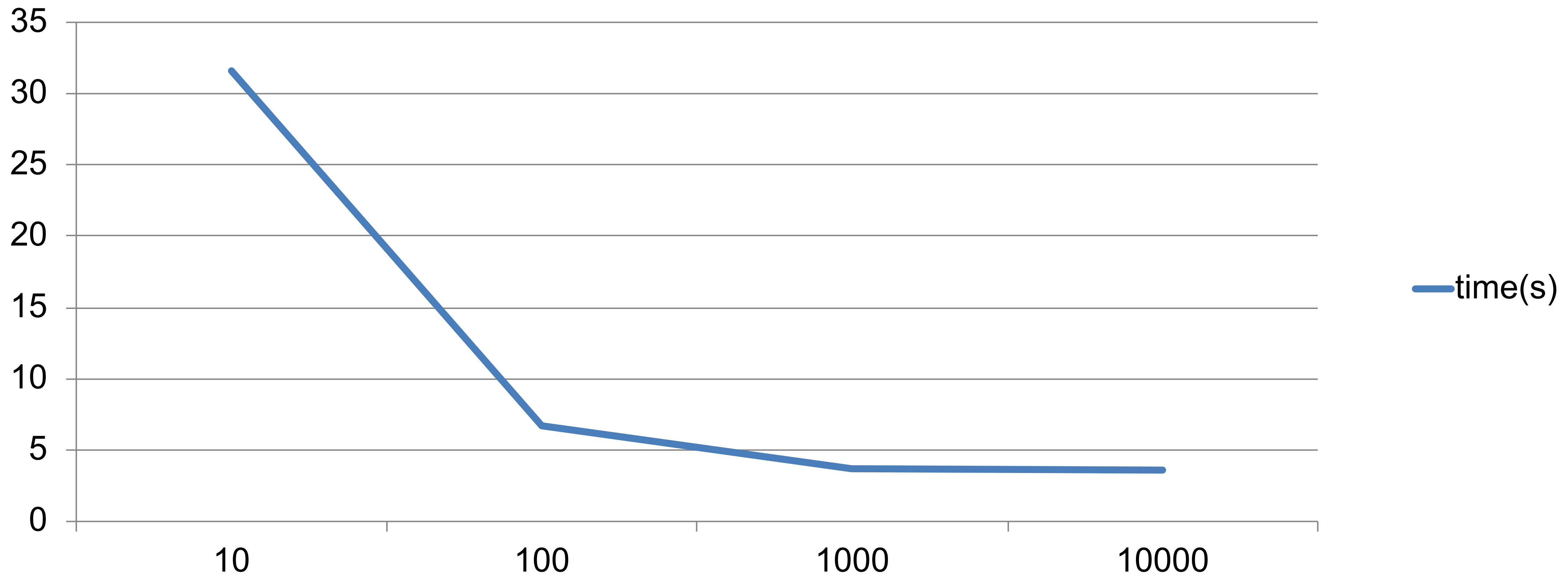
- setFetchSize
- rewriteBatchInserts

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);
```

Time it takes to fetch 1M rows



What are the options for inserting lots of data

- For each row insertExecute this is the slowest
- For each row insertBatch this would be ideal
- Insert into foo (i,j) values (1,’one’), (2,’two’) …. (n,’n’) hand rolled code
- Copy into foo from stdin…

JDBC micro benchmark suite

- Java 1.8_60
- Core i7 2.8GHz
- PostgreSQL 9.6
- 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
--------+----------------------------------
  a     | integer
  b     | character varying(100)
  c     | integer
```

INSERT Batch 1 row at a time

- For each row Insert into perf (a,b,c) values (?,?,?)
- After N rows executeBatch
- Normal mode this executes N inserts, not any faster than
- Looping over N inserts without batch mode

INSERT Batch N rows_at_a_time

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

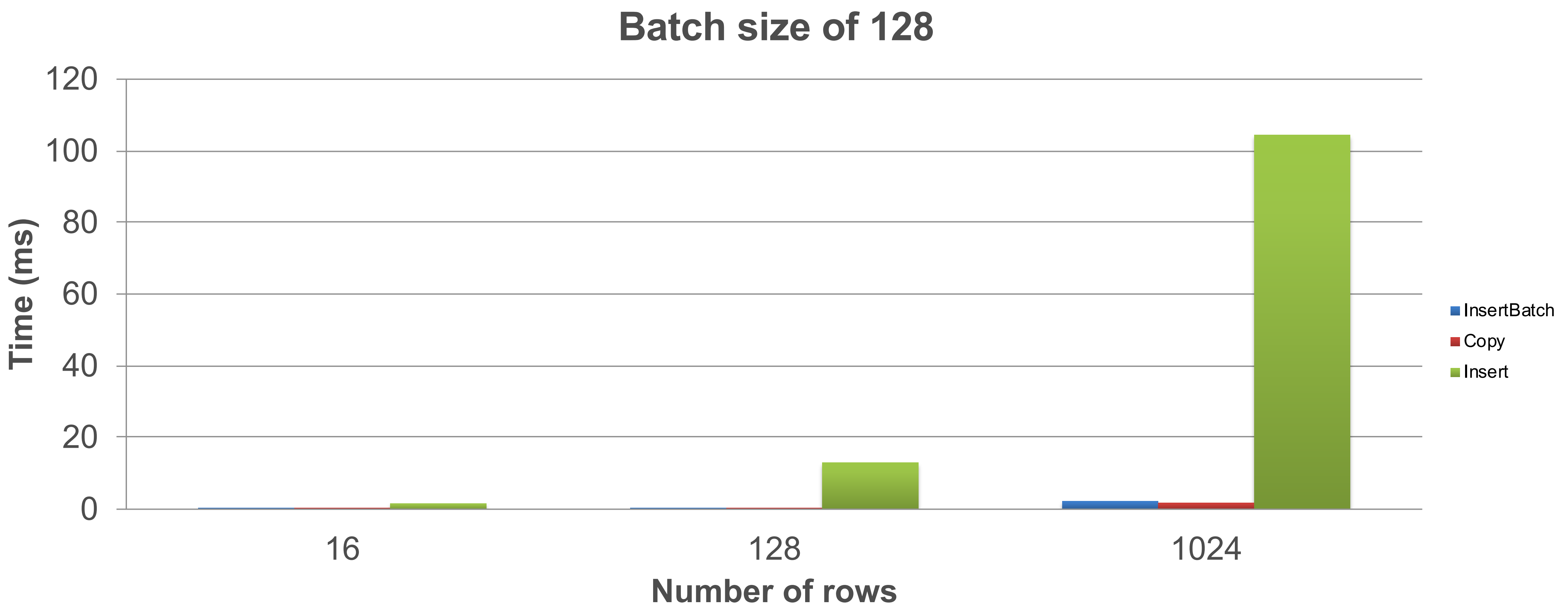
INSERT Batch with insertRewrite

- For each row Insert into perf (a,b,c) values (?,?,?)
- After N rows executeBatch
- Same as last slide except we set the connection parameter insertRewrite=true
- As of version 1209 this is has been enabled
- Same as insert into foo (i,j) values (1,'one'), (2,'two') …. (n,'n') except the driver does it for you.

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 / rows_at_a_time long
- Use the copy API to copy this into the table

Results


Batch size of 128
Time (ms)
120
100
80
60
40
20
0
16
128
1024
Number of rows
InsertBatch
Copy
Insert

Conclusion

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

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

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

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 within 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

Logical Replication Overview

- Reads the WAL logs and outputs them in any format you want
- Read changes
- Send confirmation of changes read
- GOTO read more changes

Logical Replication High level Steps

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

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);
```

Create a Logical Replication Slot

```
String outputPlugin =  ‘test_decode’;
try (PreparedStatement preparedStatement =
             connection.prepareStatement("SELECT *
FROM pg_create_logical_replication_slot(?, ?)"))
{
    preparedStatement.setString(1, slotName);
    preparedStatement.setString(2, outputPlugin);
    preparedStatement.executeQuery())

}
```

Create a replication stream

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

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());
   }
```

https://github.com/pgjdbc/pgjdbc

- Credit where credit is due:
- Much of the optimization work on the driver was done by Vladimir Sitnikov
- Rewriting batch statements thanks to Jeremy Whiting
- Replication support was provided by Vladimir Gordiychuk
- Questions ?

THANK YOU!

Dave Cramer
dave.cramer@crunchydata.ca