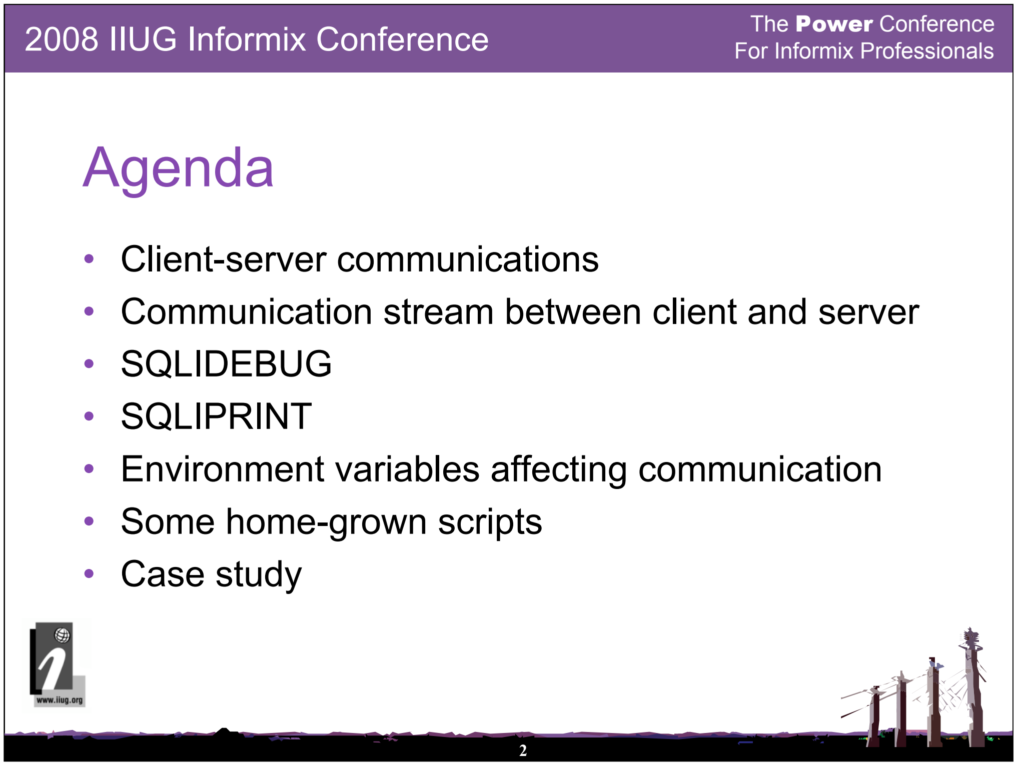
Using SQLIDEBUG to Help Streamline Your Application

Mike Lowe
IBM Data Management

Session Code: D15
Wednesday, April 30, 2008 • 13:00 – 14:00

2008 IIUG Informix Conference

www.iiug.org



Agenda

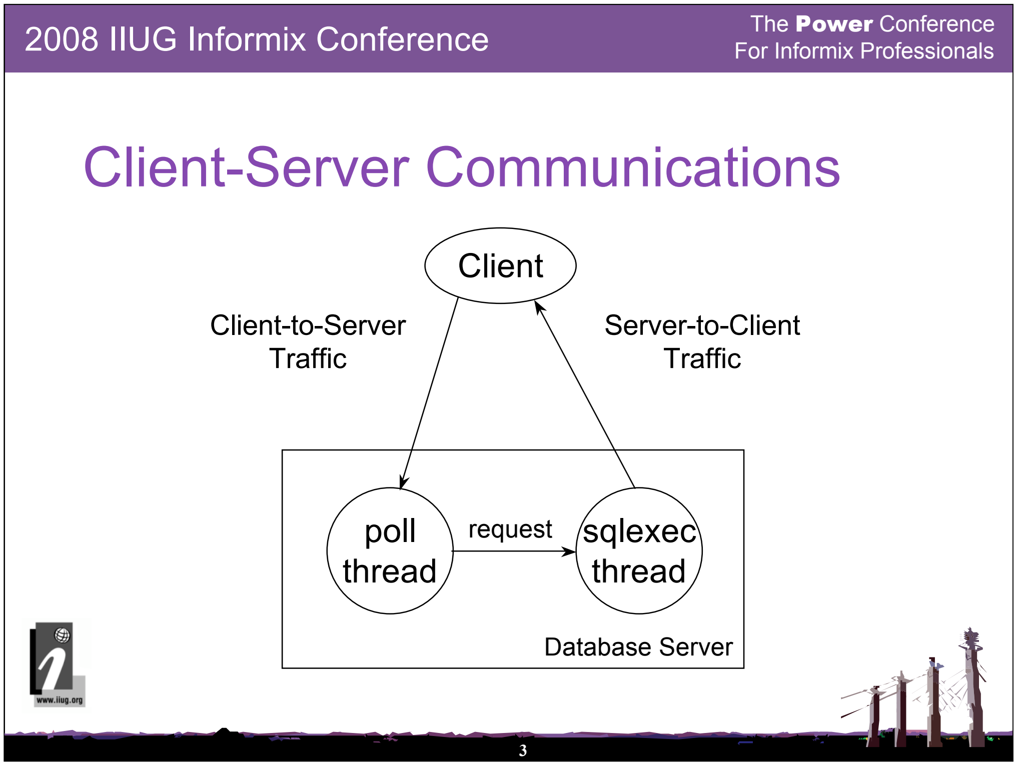
- Client-server communications
- Communication stream between client and server
- SQLIDEBUG
- SQLIPRINT
- Environment variables affecting communication
- Some home-grown scripts
- Case study

In this session we will cover some basic concepts of the client-server communication process, then delve deeper into the details of those communications.

We will cover the built-in SQLIDEBUG feature and how to use it, then discuss the companion tool SQLIPRINT.

We will also then discuss some of the environment variables which can be used to affect the efficiency of client-server communications.

Finally, we will look at some home-grown scripts which will help you understand exactly what is being sent to the server from the client, and look at a case study where these scripts were used on a client application.



Client-Server Communications

Client

Client-to-Server
Traffic

Server-to-Client
Traffic

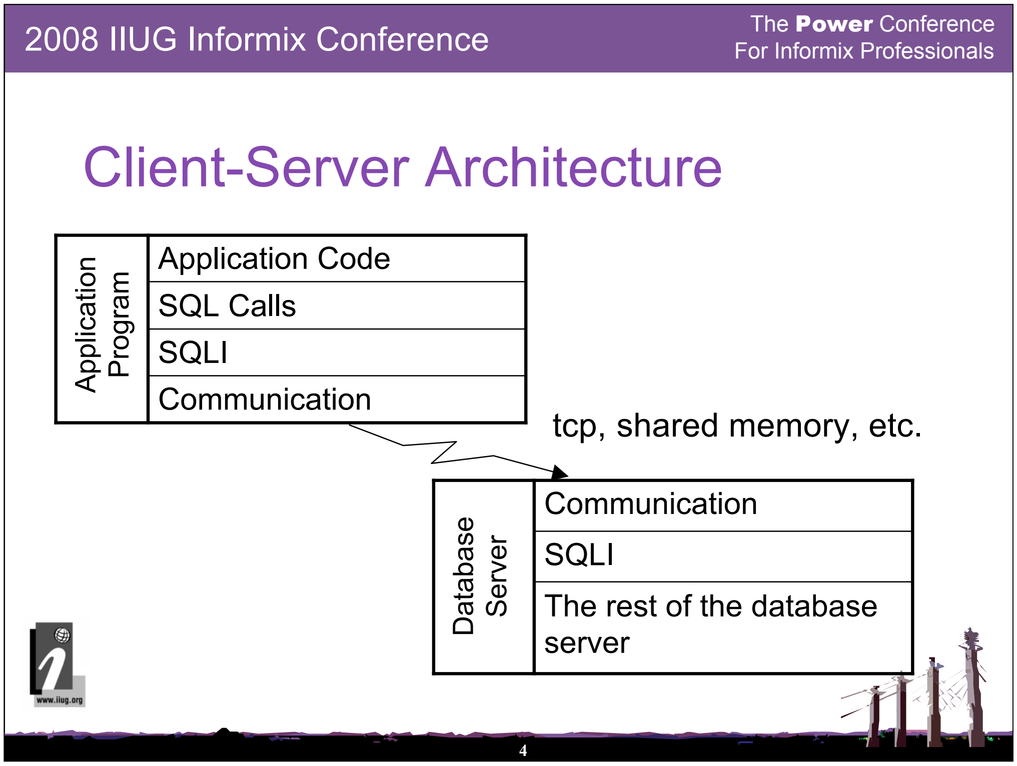
poll
thread

request

sqlexec
thread

Database Server

This slide depicts the flow of client-server communications traffic.



Client-Server Architecture

Application Program

Application Code

SQL Calls

SQLI

Communication

tcp, shared memory, etc.

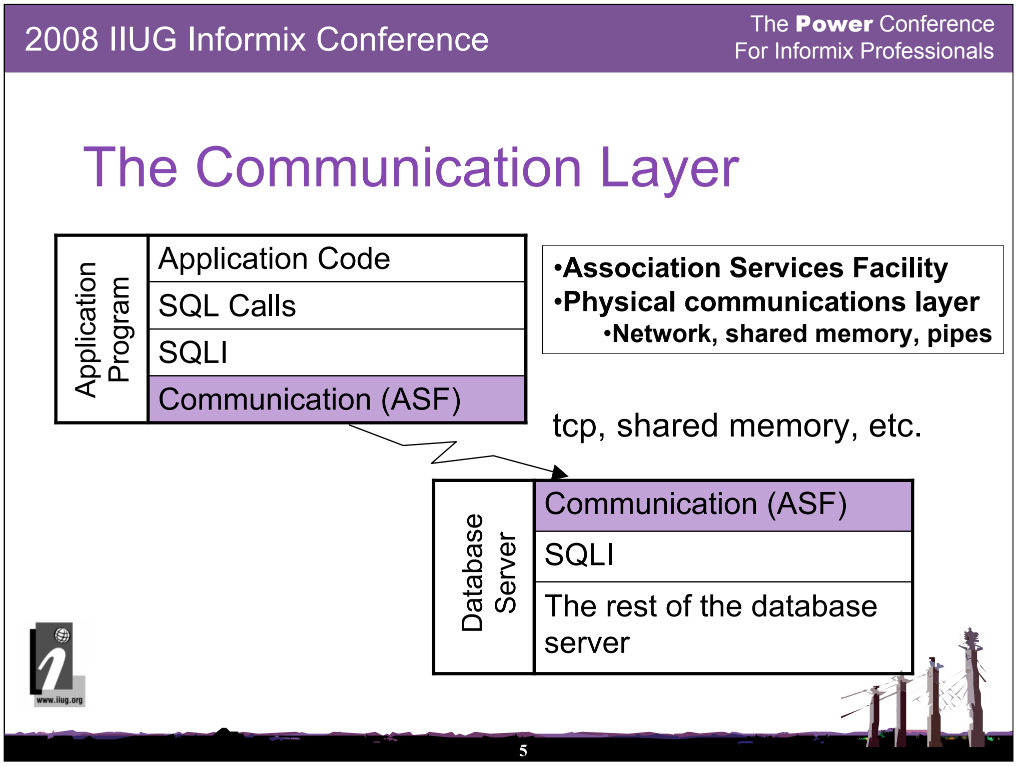
Database Server

Communication

SQLI

The rest of the database server

The overall architecture looks something like what is depicted on this slide.



2008 IIUG Informix Conference

The **Power** Conference
For Informix Professionals

The Communication Layer

Application Program

Application Code

SQL Calls

SQLI

Communication (ASF)

- **Association Services Facility**
- **Physical communications layer**
  - **Network, shared memory, pipes**

tcp, shared memory, etc.

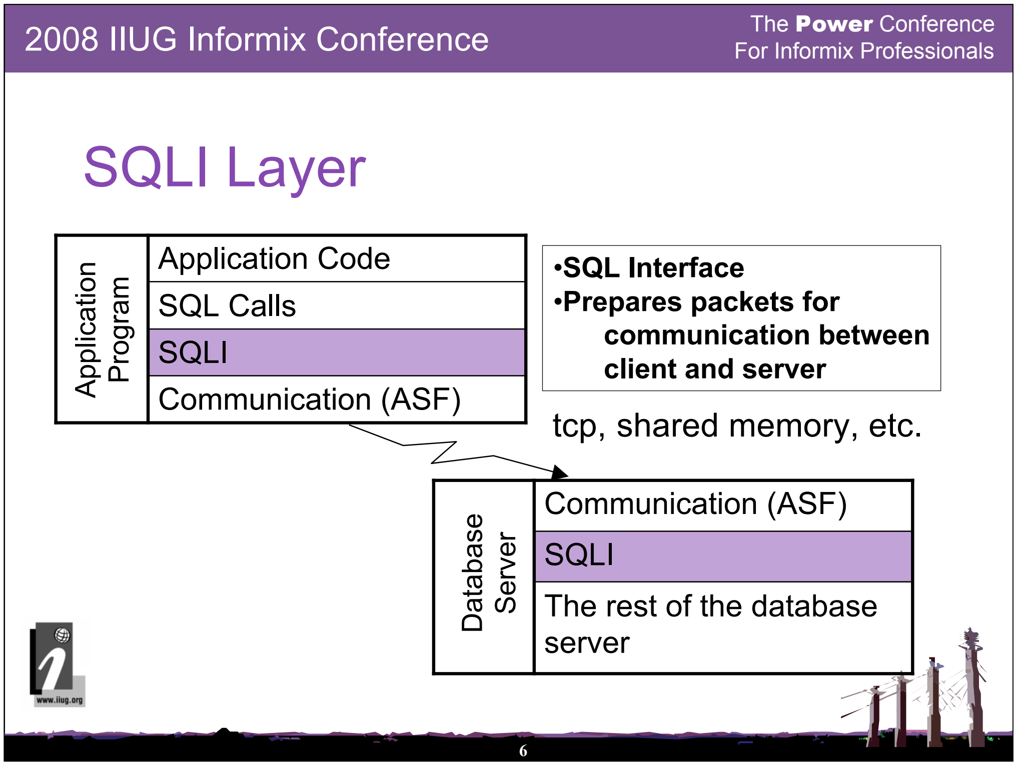
Database Server

Communication (ASF)

SQLI

The rest of the database server

The Communication layer is the physical communication layer, consisting of whatever components are necessary for the type of communication protocol involved – TCP/IP, shared memory, stream pipes, etc.

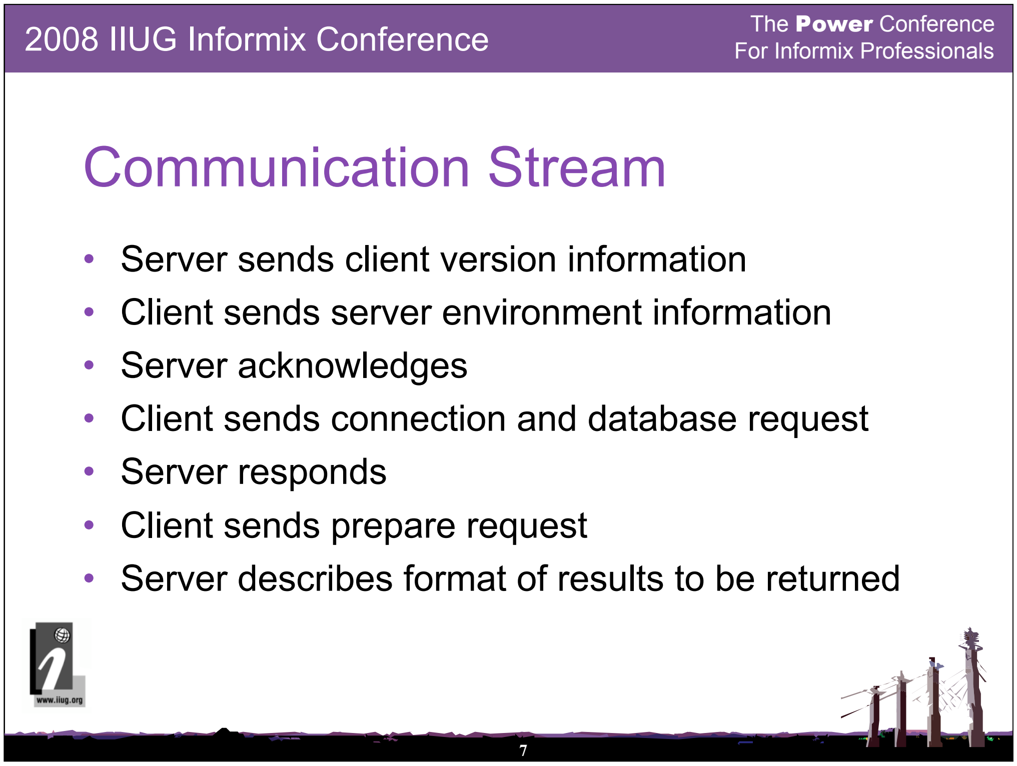


SQLI Layer

- **SQL Interface**
- **Prepares packets for communication between client and server**

The SQLI layer, or SQL Interface layer, is where we will focus in this session.

The SQLI layer takes the SQL calls from the application code and prepares packets for communication between the client and the server.



2008 IIUG Informix Conference

The **Power** Conference
For Informix Professionals

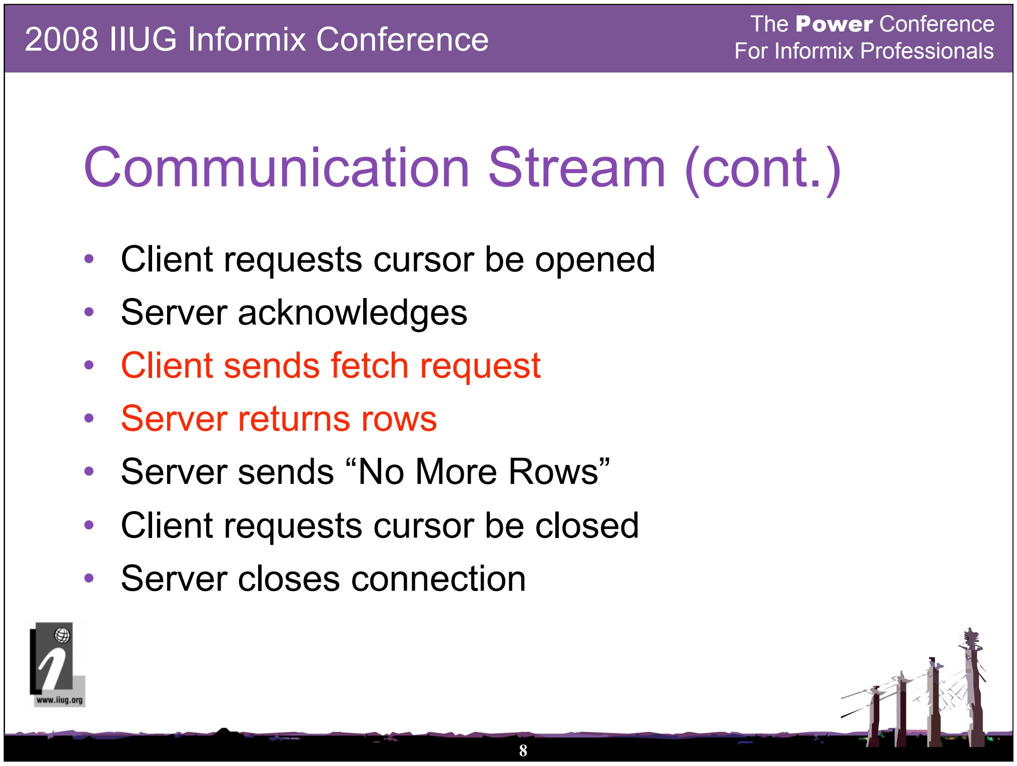
Communication Stream

- Server sends client version information
- Client sends server environment information
- Server acknowledges
- Client sends connection and database request
- Server responds
- Client sends prepare request
- Server describes format of results to be returned

www.iiug.org

The communication stream consists of a lot of message traffic between the client and the server.

The typical communication stream looks something like what is depicted in this and the following slide.

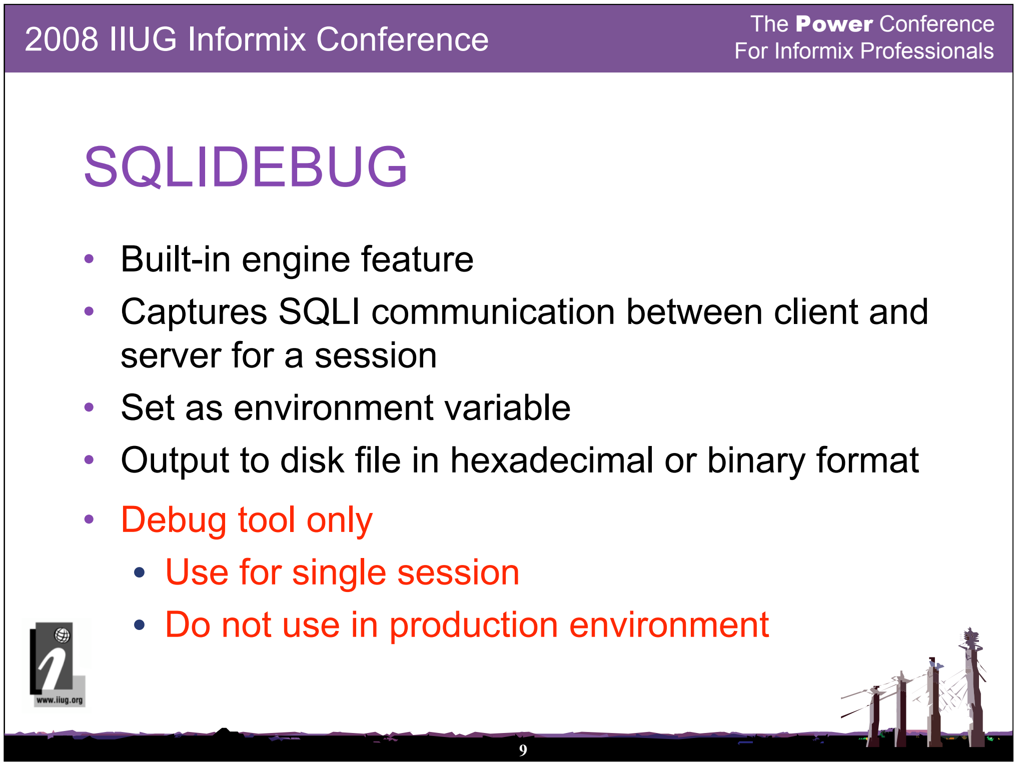


Communication Stream (cont.)

- Client requests cursor be opened
- Server acknowledges
- Client sends fetch request
- Server returns rows
- Server sends “No More Rows”
- Client requests cursor be closed
- Server closes connection

www.iiug.org

Most of the message traffic consists of one message and one response.

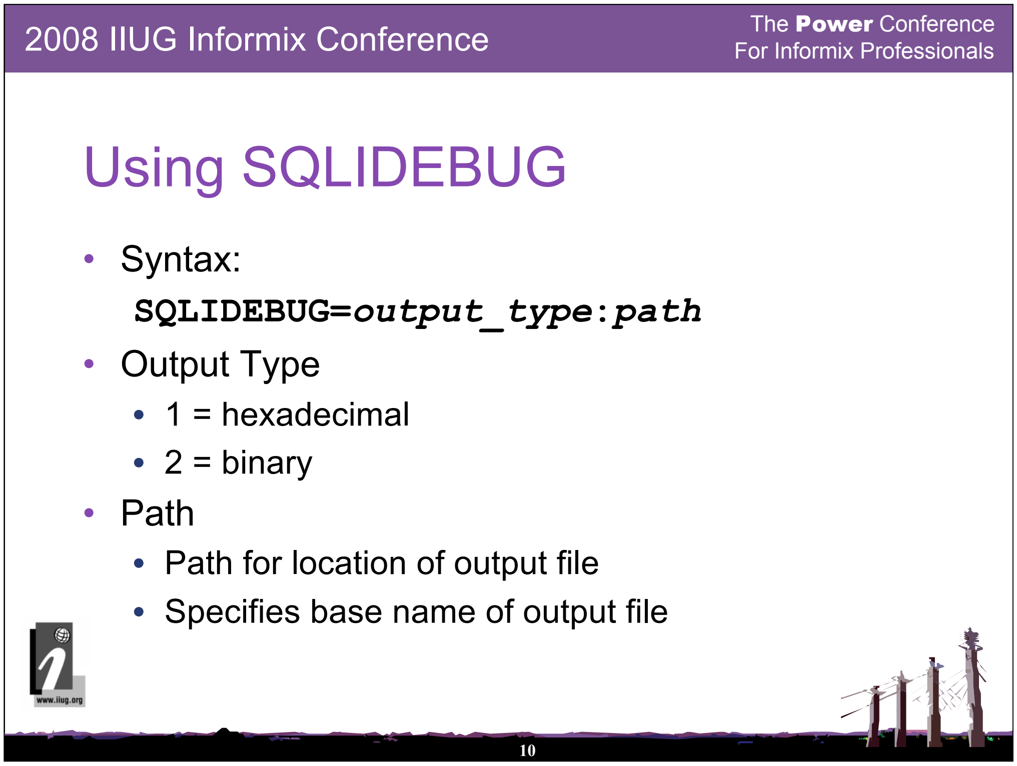


SQLIDEBUG

- Built-in engine feature
- Captures SQLI communication between client and server for a session
- Set as environment variable
- Output to disk file in hexadecimal or binary format
- Debug tool only
  - Use for single session
  - Do not use in production environment

SQLIDEBUG is a built-in engine feature that lets us capture the SQLI layer communications between a client and a server.

This capture is enabled by setting the SQLIDEBUG environment variable.



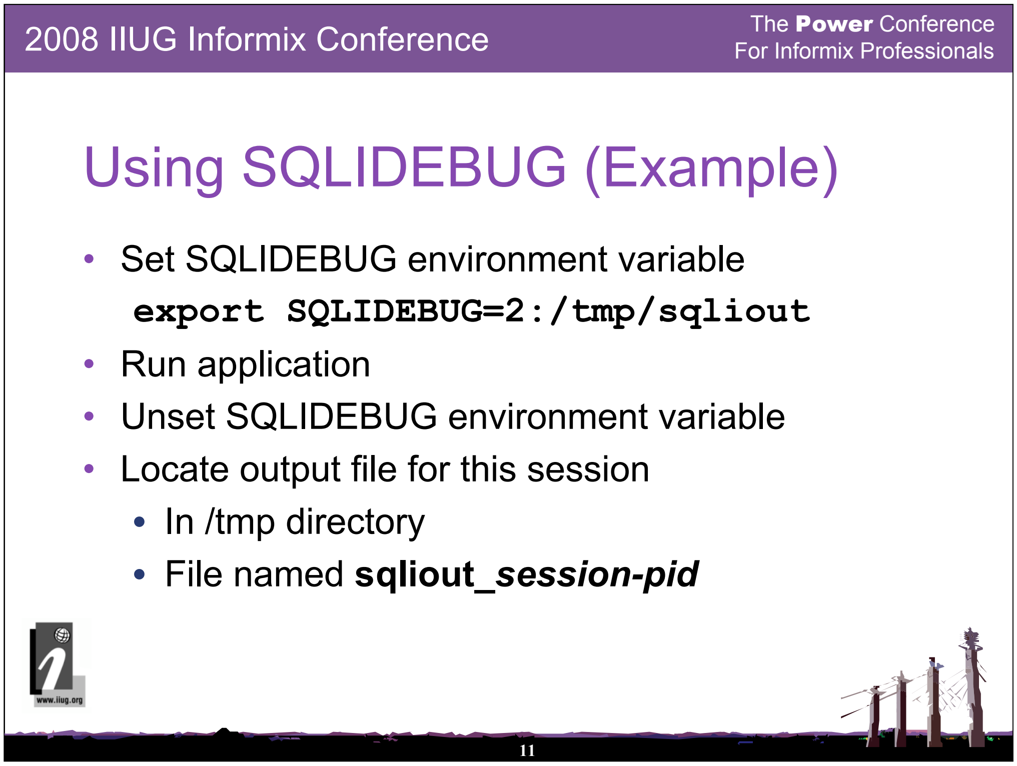
Using SQLIDEBUG

- Syntax:

  SQLIDEBUG=*output_type*:*path*

- Output Type
  - 1 = hexadecimal
  - 2 = binary
- Path
  - Path for location of output file
  - Specifies base name of output file

This slide describes the syntax for setting SQLIDEBUG.



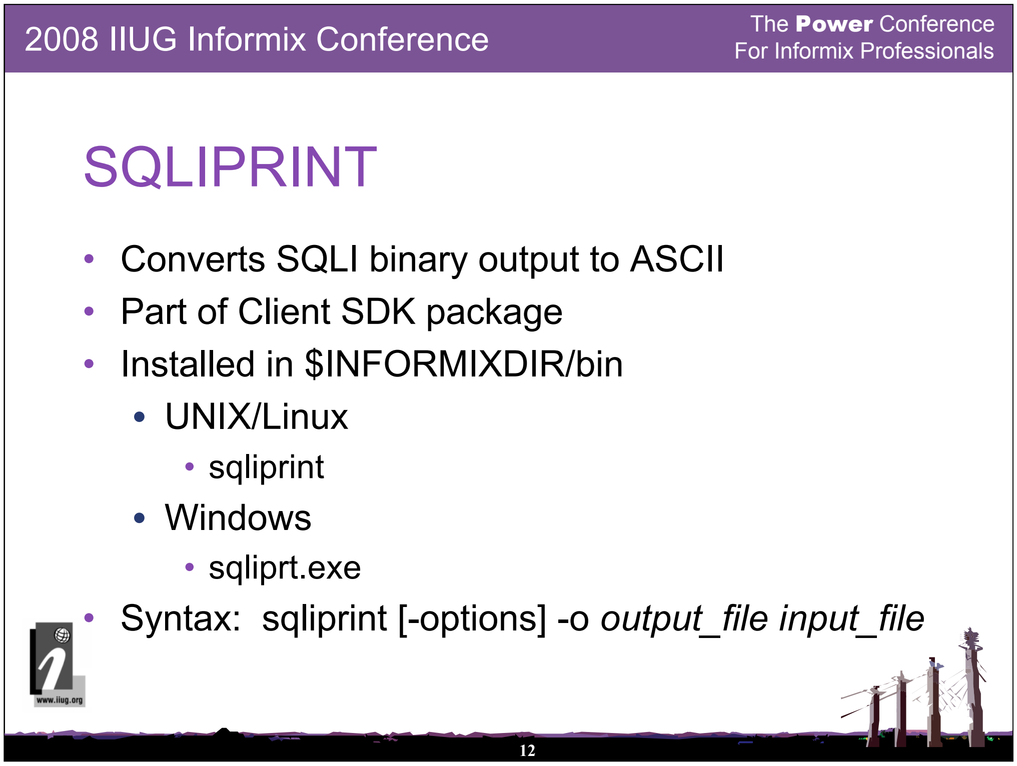
Using SQLIDEBUG (Example)

- Set SQLIDEBUG environment variable

```
export SQLIDEBUG=2:/tmp/sqliout
```

- Run application
- Unset SQLIDEBUG environment variable
- Locate output file for this session
  - In /tmp directory
  - File named **sqliout_*session-pid***

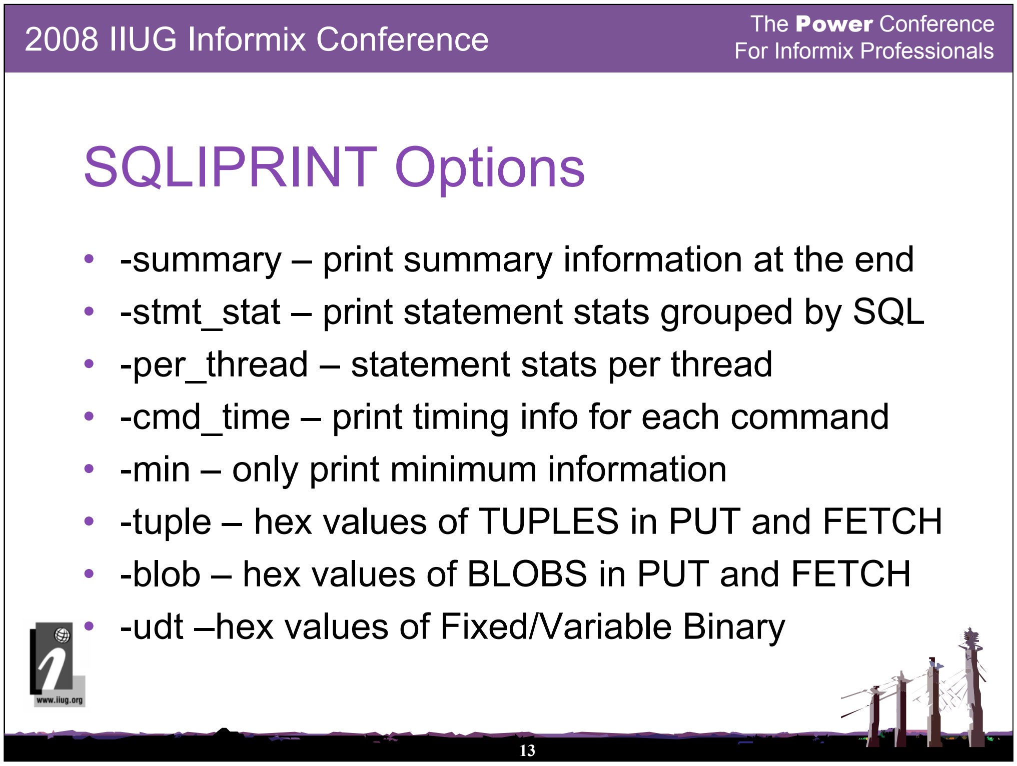
To use SQLIDEBUG, set the SQLIDEBUG environment variable as shown in the slide, run the application, and then unset SQLIDEBUG.



SQLIPRINT

- Converts SQLI binary output to ASCII
- Part of Client SDK package
- Installed in $INFORMIXDIR/bin
  - UNIX/Linux
    - sqliprint
  - Windows
    - sqliprt.exe
- Syntax: sqliprint [-options] -o *output_file input_file*

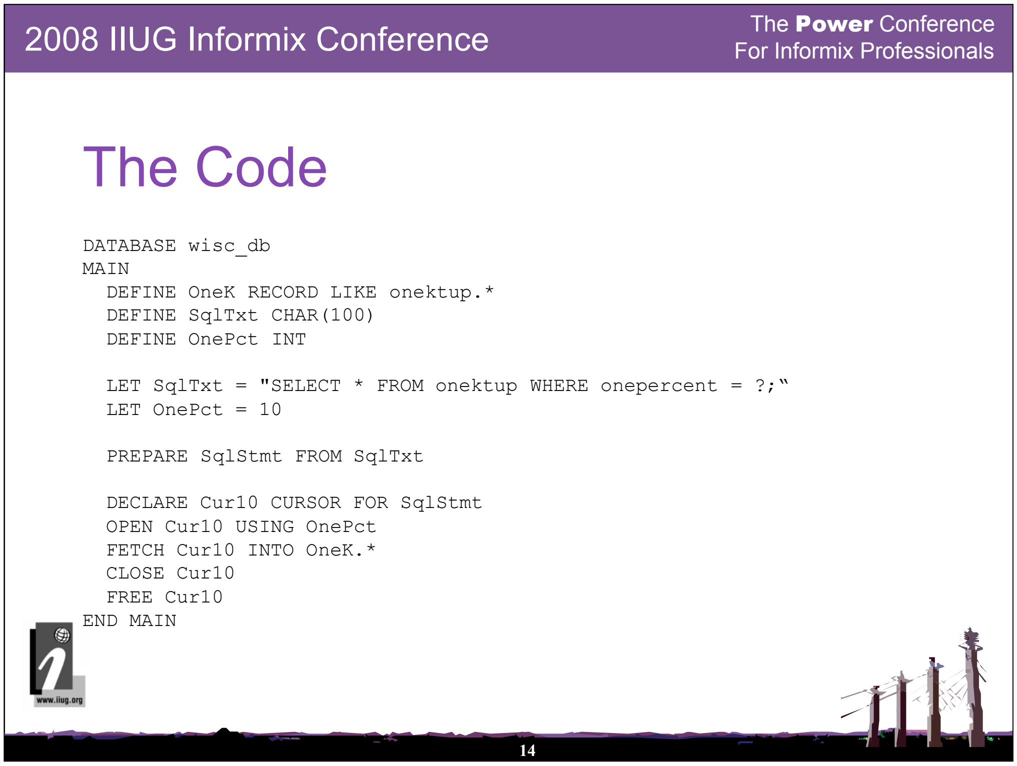
SQLIPRINT is the tool that converts the binary output from SQLIDEBUG to ASCII. It is part of the Client CSDK package.



SQLIPRINT Options

- -summary – print summary information at the end
- -stmt_stat – print statement stats grouped by SQL
- -per_thread – statement stats per thread
- -cmd_time – print timing info for each command
- -min – only print minimum information
- -tuple – hex values of TUPLES in PUT and FETCH
- -blob – hex values of BLOBS in PUT and FETCH
- -udt –hex values of Fixed/Variable Binary

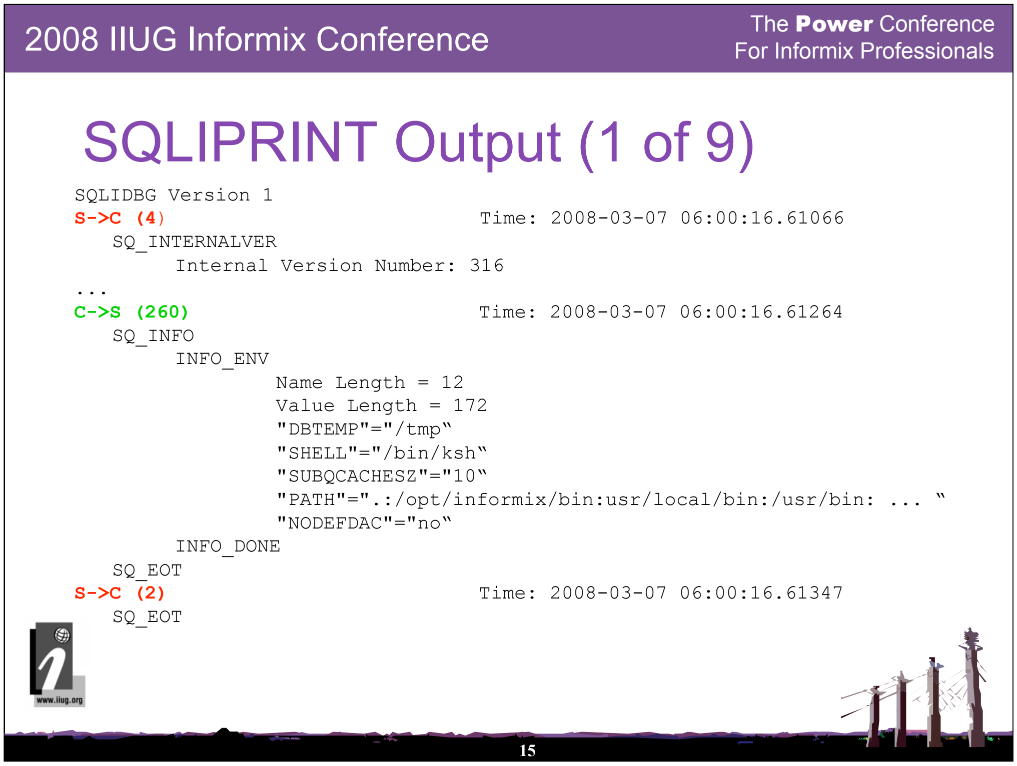
This slide lists some of the options available with sqliprint.



The Code

```
DATABASE wisc_db
MAIN
  DEFINE OneK RECORD LIKE onektup.*
  DEFINE SqlTxt CHAR(100)
  DEFINE OnePct INT

  LET SqlTxt = "SELECT * FROM onektup WHERE onepercent = ?;"
  LET OnePct = 10

  PREPARE SqlStmt FROM SqlTxt

  DECLARE Cur10 CURSOR FOR SqlStmt
  OPEN Cur10 USING OnePct
  FETCH Cur10 INTO OneK.*
  CLOSE Cur10
  FREE Cur10
END MAIN
```

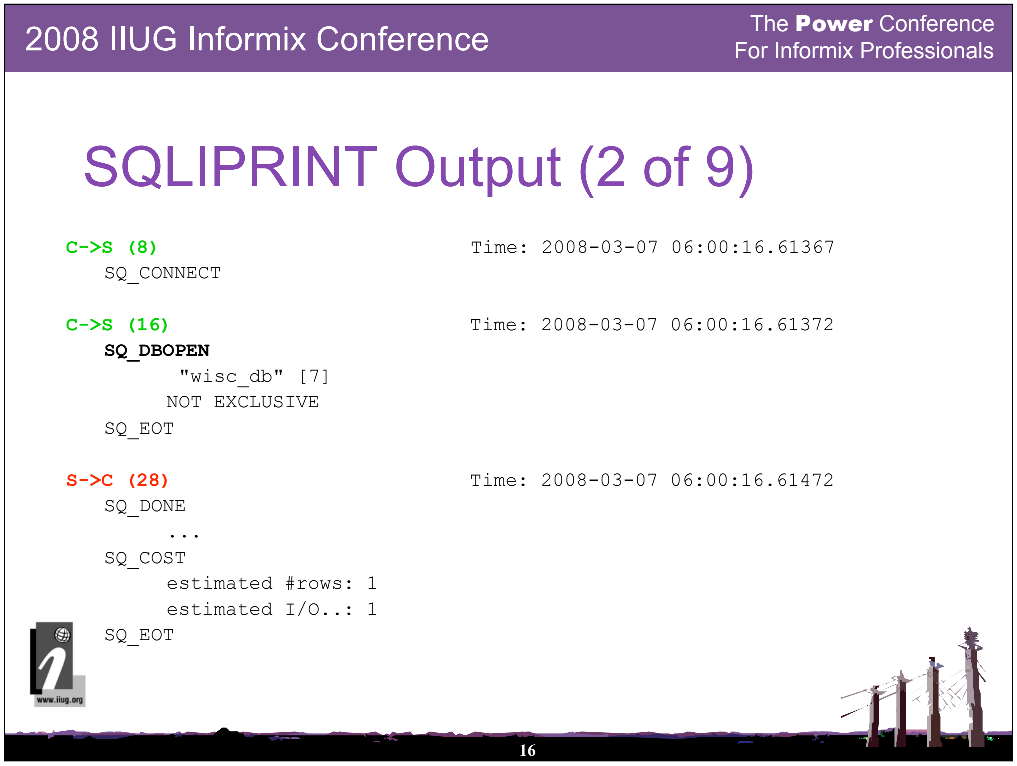
This slide shows a very simple 4GL program that we will use to explore the SQLI communications stream in the following slides.



SQLIPRINT Output (1 of 9)

```
SQLIDBG Version 1
S->C (4)                           Time: 2008-03-07 06:00:16.61066
   SQ_INTERNALVER
         Internal Version Number: 316
...
C->S (260)                         Time: 2008-03-07 06:00:16.61264
   SQ_INFO
         INFO_ENV
                 Name Length = 12
                 Value Length = 172
                 "DBTEMP"="/tmp‟
                 "SHELL"="/bin/ksh‟
                 "SUBQCACHESZ"="10‟
                 "PATH"=".:/opt/informix/bin:usr/local/bin:/usr/bin: ... ‟
                 "NODEFDAC"="no‟
         INFO_DONE
   SQ_EOT
S->C (2)                           Time: 2008-03-07 06:00:16.61347
   SQ_EOT
```

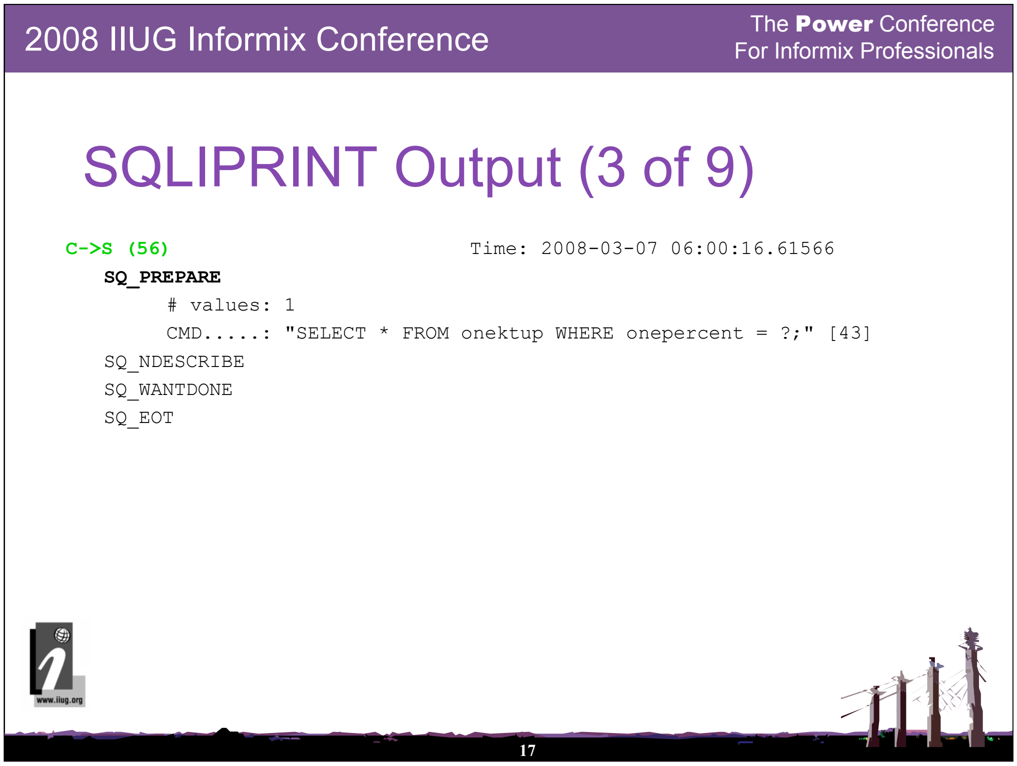
This is the first part of the output from sqliprint.



SQLIPRINT Output (2 of 9)

```
C->S (8)                         Time: 2008-03-07 06:00:16.61367
   SQ_CONNECT

C->S (16)                        Time: 2008-03-07 06:00:16.61372
   SQ_DBOPEN
         "wisc_db" [7]
        NOT EXCLUSIVE
   SQ_EOT

S->C (28)                        Time: 2008-03-07 06:00:16.61472
   SQ_DONE
        ...
   SQ_COST
        estimated #rows: 1
        estimated I/O..: 1
   SQ_EOT
```

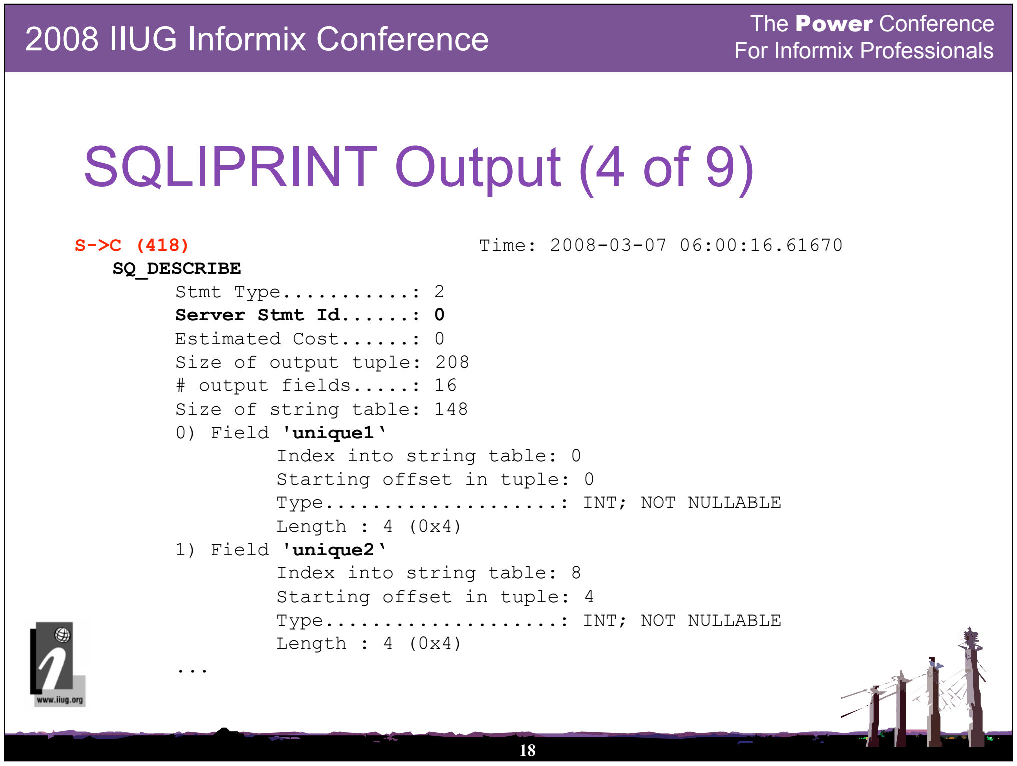
Next we get a connect and open request for the wisc_db database, and the server’s acknowledgment.



SQLIPRINT Output (3 of 9)

```
C->S (56)                         Time: 2008-03-07 06:00:16.61566
    SQ_PREPARE
         # values: 1
         CMD.....: "SELECT * FROM onektup WHERE onepercent = ?;" [43]
    SQ_NDESCRIBE
    SQ_WANTDONE
    SQ_EOT
```

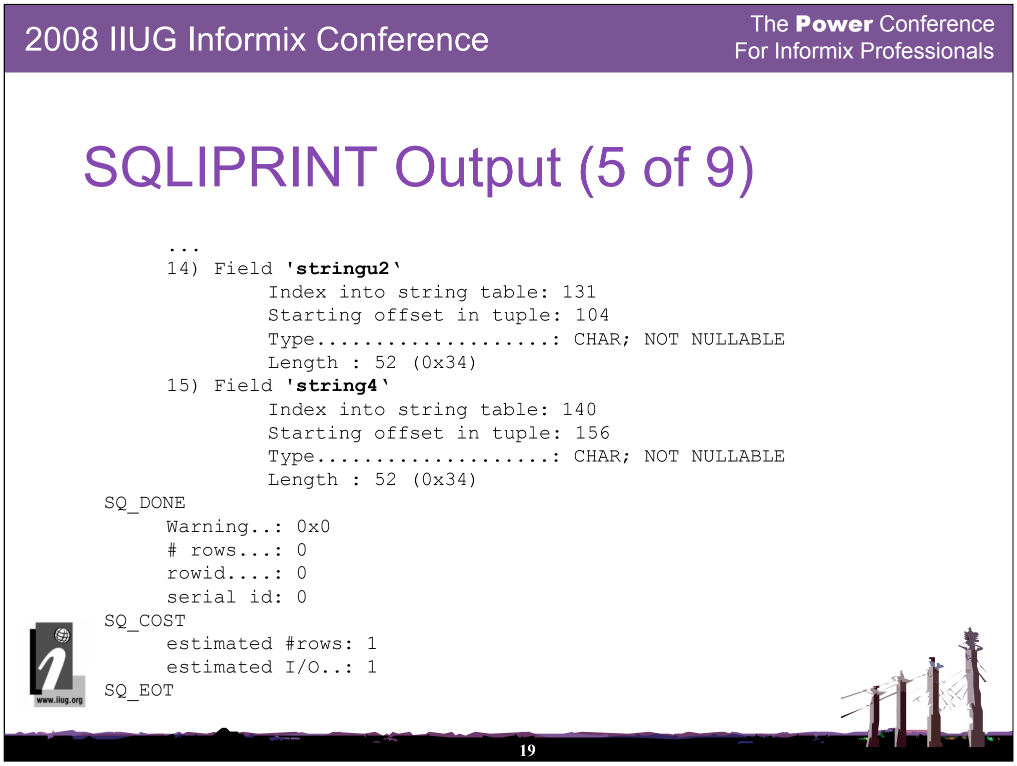
Then the client sends a request to prepare the SQL select statement.



SQLIPRINT Output (4 of 9)

```
S->C (418)                        Time: 2008-03-07 06:00:16.61670
   SQ_DESCRIBE
        Stmt Type...........: 2
        Server Stmt Id......: 0
        Estimated Cost......: 0
        Size of output tuple: 208
        # output fields.....: 16
        Size of string table: 148
        0) Field 'unique1`
                Index into string table: 0
                Starting offset in tuple: 0
                Type....................: INT; NOT NULLABLE
                Length : 4 (0x4)
        1) Field 'unique2`
                Index into string table: 8
                Starting offset in tuple: 4
                Type....................: INT; NOT NULLABLE
                Length : 4 (0x4)
        ...
```

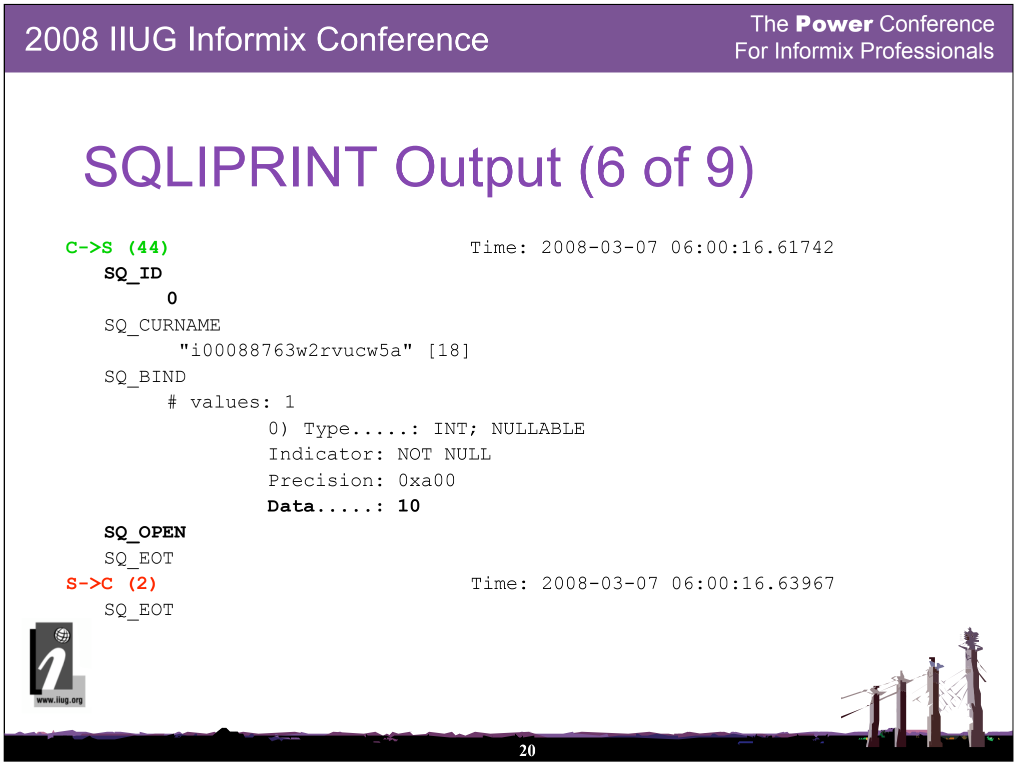
Now the server **describes** the data stream that will be returned from the prepared SQL statement (SQ_DESCRIBE).



SQLIPRINT Output (5 of 9)

```
     ...
     14) Field 'stringu2'
              Index into string table: 131
              Starting offset in tuple: 104
              Type....................: CHAR; NOT NULLABLE
              Length : 52 (0x34)
     15) Field 'string4'
              Index into string table: 140
              Starting offset in tuple: 156
              Type....................: CHAR; NOT NULLABLE
              Length : 52 (0x34)
SQ_DONE
     Warning..: 0x0
     # rows...: 0
     rowid....: 0
     serial id: 0
SQ_COST
     estimated #rows: 1
     estimated I/O..: 1
SQ_EOT
```

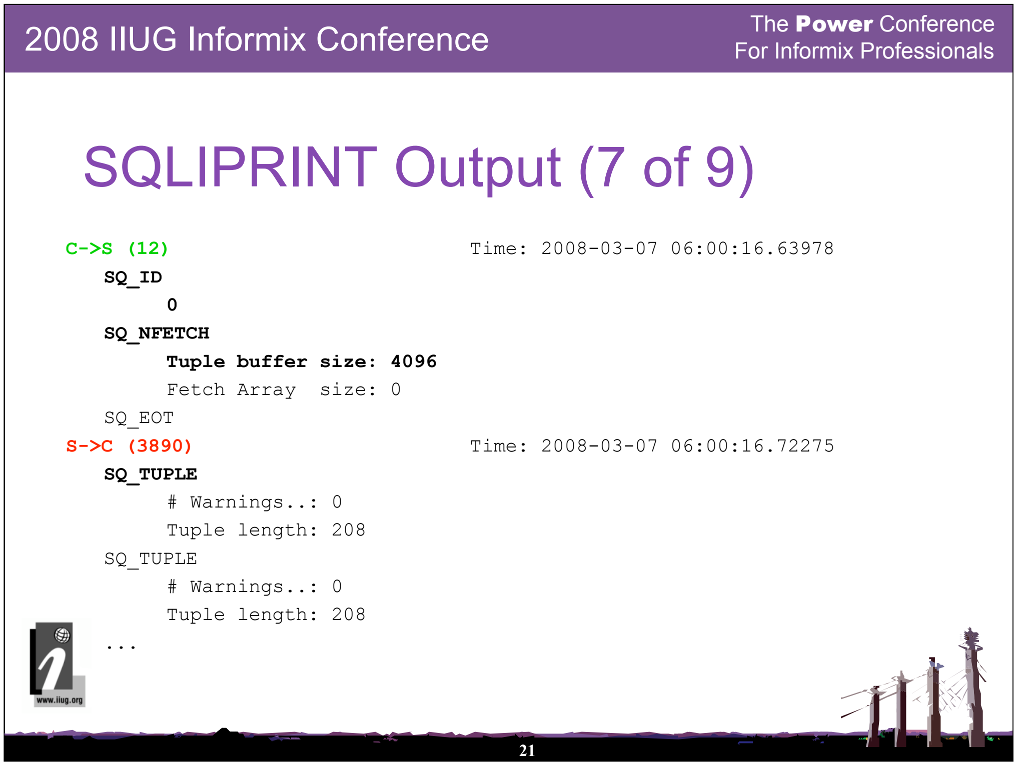
This slide shows the end of the SQ_DESCRIBE section sent from the server to the client.



SQLIPRINT Output (6 of 9)

```
C->S (44)                       Time: 2008-03-07 06:00:16.61742
   SQ_ID
        0
   SQ_CURNAME
         "i00088763w2rvucw5a" [18]
   SQ_BIND
        # values: 1
                0) Type.....: INT; NULLABLE
                Indicator: NOT NULL
                Precision: 0xa00
                Data.....: 10
   SQ_OPEN
   SQ_EOT
S->C (2)                        Time: 2008-03-07 06:00:16.63967
   SQ_EOT
```

Now the client sends the variable data for the SQL statement to the server.

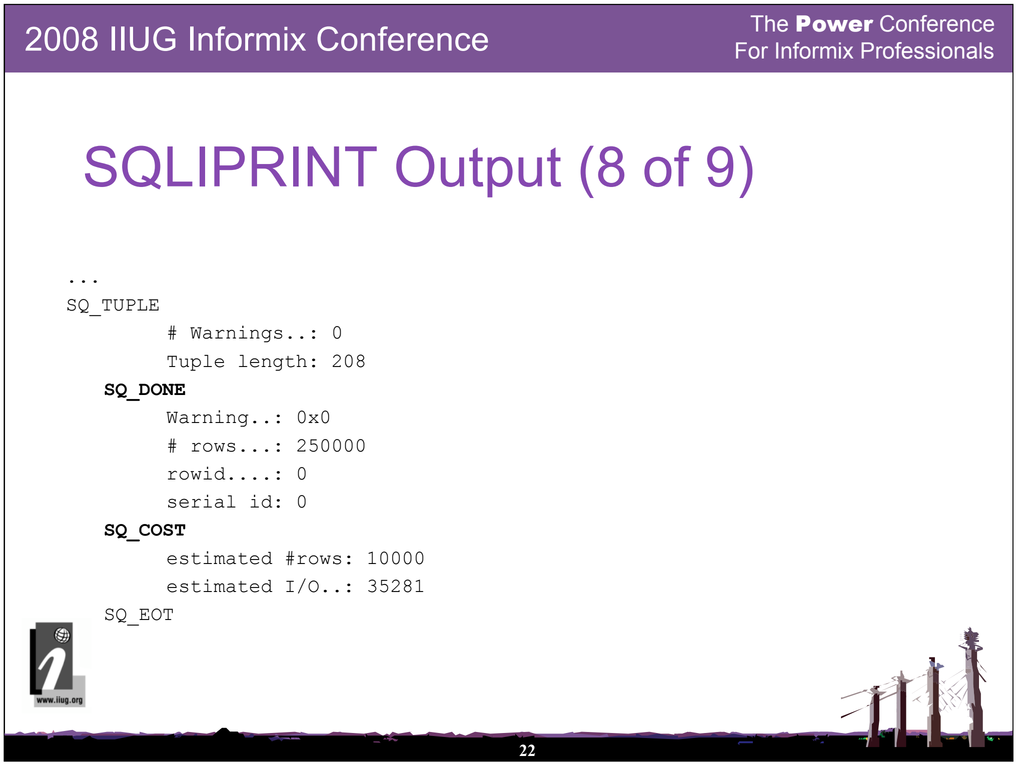


SQLIPRINT Output (7 of 9)

```
C->S (12)                        Time: 2008-03-07 06:00:16.63978
   SQ_ID
        0
   SQ_NFETCH
        Tuple buffer size: 4096
        Fetch Array  size: 0
   SQ_EOT
S->C (3890)                      Time: 2008-03-07 06:00:16.72275
   SQ_TUPLE
        # Warnings..: 0
        Tuple length: 208
   SQ_TUPLE
        # Warnings..: 0
        Tuple length: 208
   ...
```

www.iiug.org

Once the cursor is opened, we tell the server we want it to start returning our data.

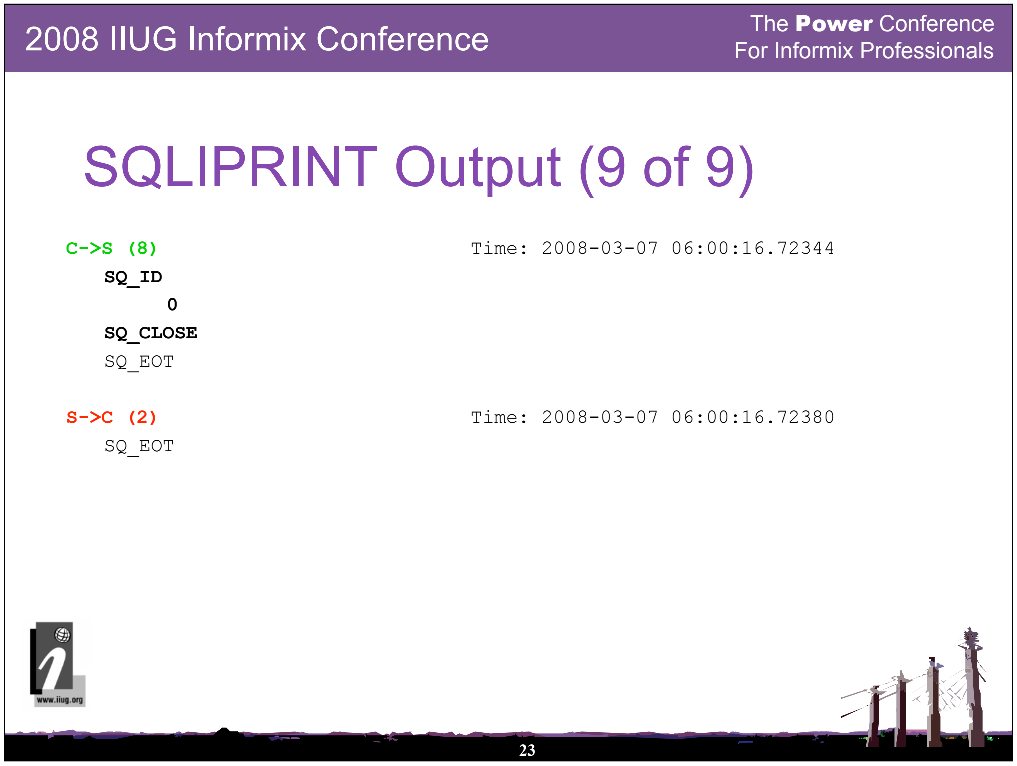


SQLIPRINT Output (8 of 9)

```
...
SQ_TUPLE
        # Warnings..: 0
        Tuple length: 208
   SQ_DONE
        Warning..: 0x0
        # rows...: 250000
        rowid....: 0
        serial id: 0
   SQ_COST
        estimated #rows: 10000
        estimated I/O..: 35281
   SQ_EOT
```

www.iiug.org

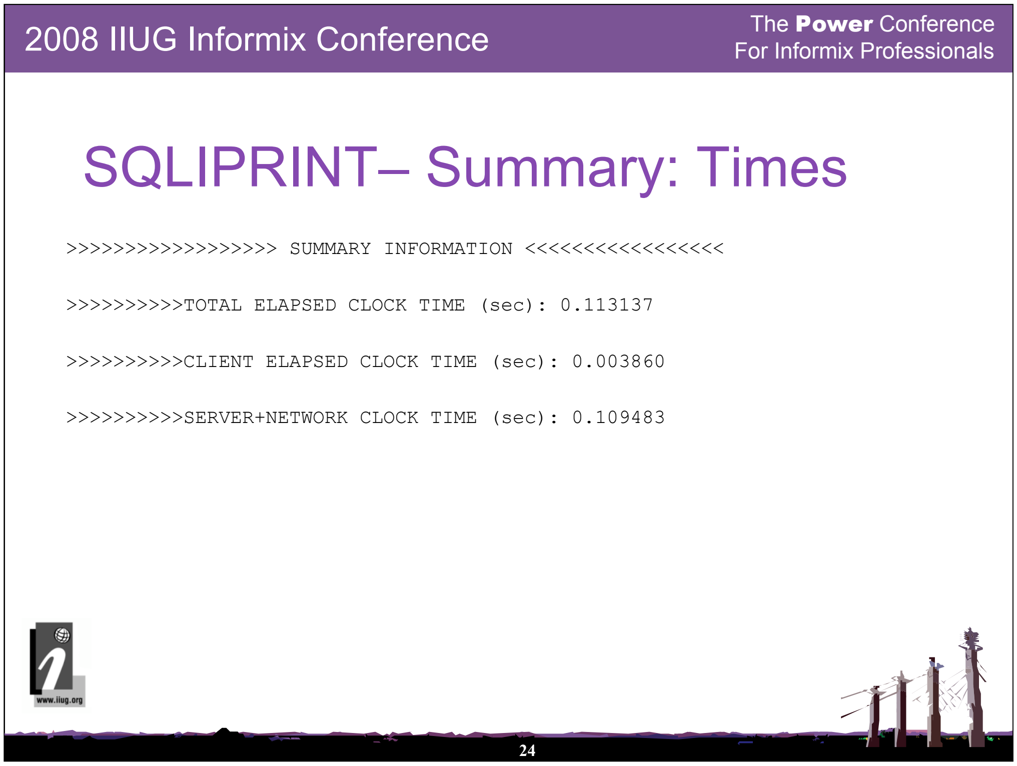
Here we see that the server has reached the end of the data, and sends the SQ_DONE message to the client.



SQLIPRINT Output (9 of 9)

```
C->S (8)                         Time: 2008-03-07 06:00:16.72344
   SQ_ID
        0
   SQ_CLOSE
   SQ_EOT

S->C (2)                         Time: 2008-03-07 06:00:16.72380
   SQ_EOT
```

Finally, we close the cursor, identifying it by its statement ID of 0.

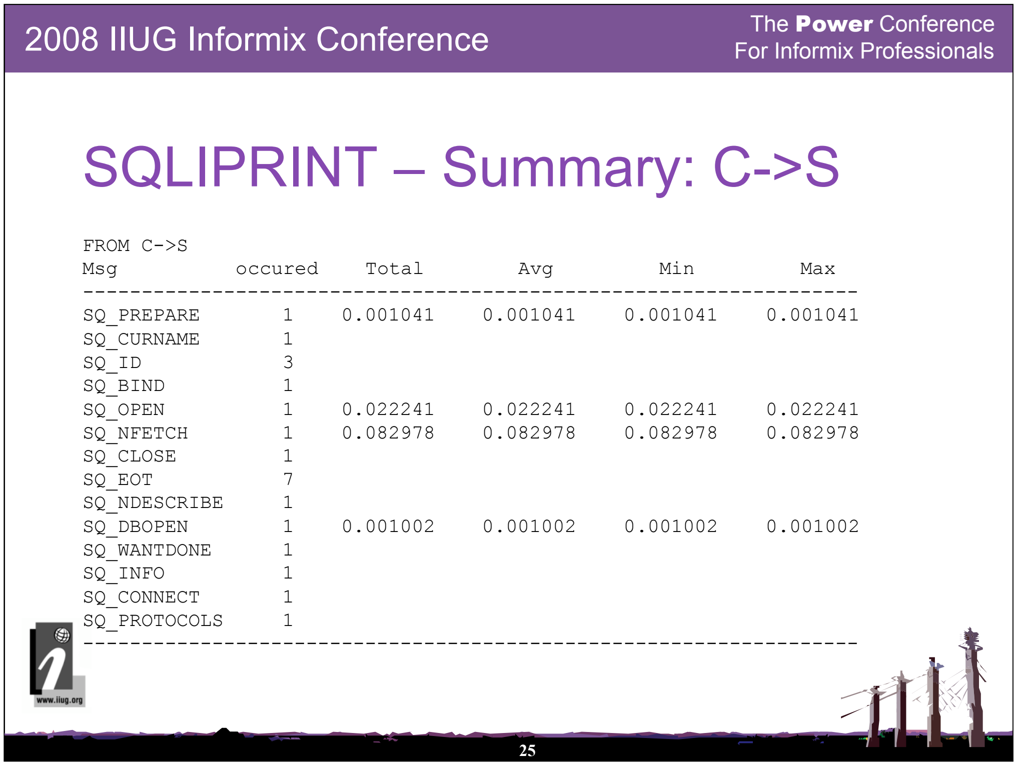


SQLIPRINT– Summary: Times

```
>>>>>>>>>>>>>>>>>> SUMMARY INFORMATION <<<<<<<<<<<<<<<<<

>>>>>>>>>>TOTAL ELAPSED CLOCK TIME (sec): 0.113137

>>>>>>>>>>CLIENT ELAPSED CLOCK TIME (sec): 0.003860

>>>>>>>>>>SERVER+NETWORK CLOCK TIME (sec): 0.109483
```

www.iiug.org

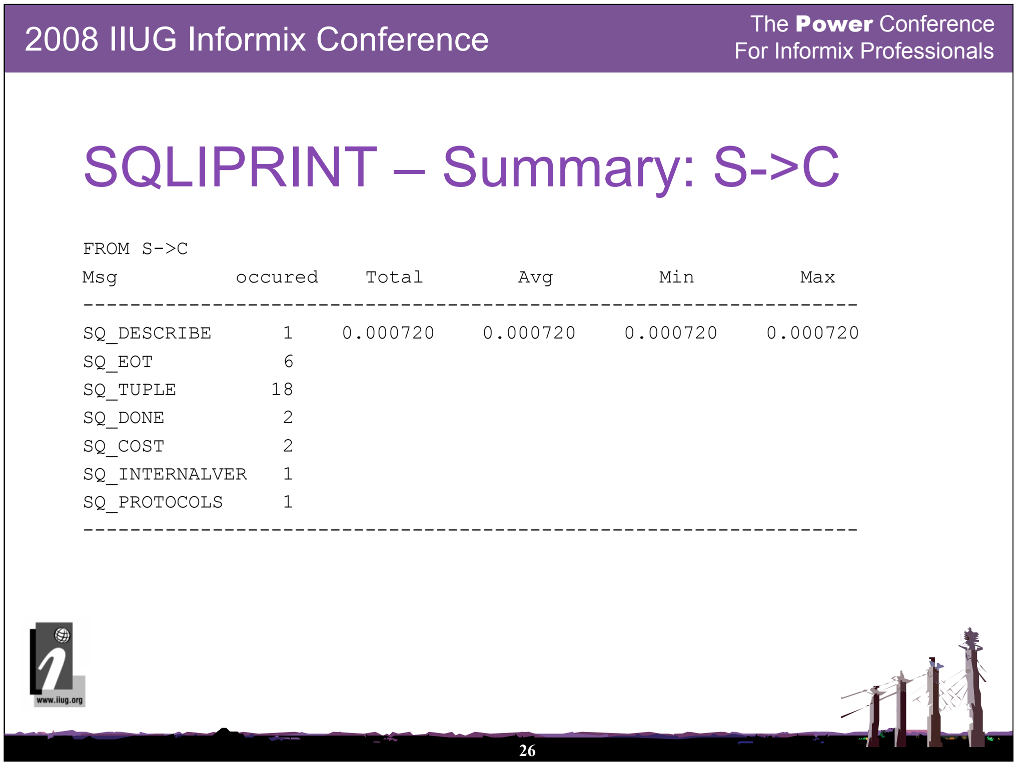
By using the “-summary” option, we get a synopsis of communications traffic listed at the bottom of the output.



SQLIPRINT – Summary: C->S

```
FROM C->S
Msg              occured     Total         Avg          Min          Max
-------------------------------------------------------------------
SQ_PREPARE         1     0.001041     0.001041     0.001041     0.001041
SQ_CURNAME         1
SQ_ID              3
SQ_BIND            1
SQ_OPEN            1     0.022241     0.022241     0.022241     0.022241
SQ_NFETCH          1     0.082978     0.082978     0.082978     0.082978
SQ_CLOSE           1
SQ_EOT             7
SQ_NDESCRIBE       1
SQ_DBOPEN          1     0.001002     0.001002     0.001002     0.001002
SQ_WANTDONE        1
SQ_INFO            1
SQ_CONNECT         1
SQ_PROTOCOLS       1
-------------------------------------------------------------------
```

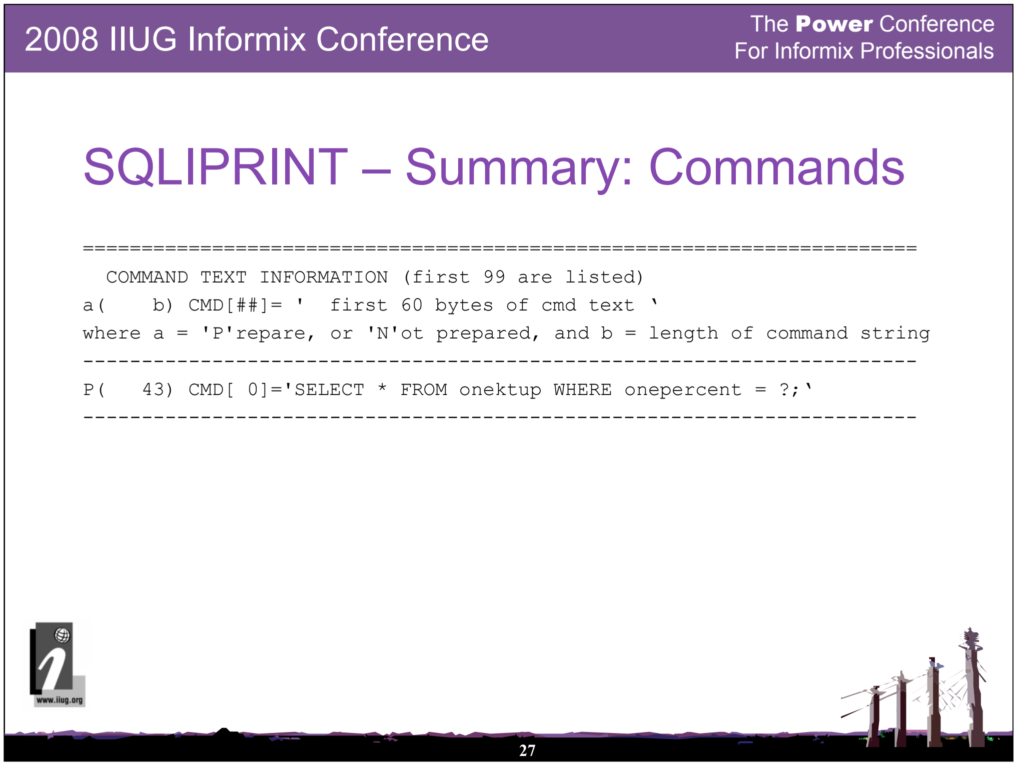
The second section is a summary of the messages sent from the client to the server.



SQLIPRINT – Summary: S->C

```
FROM S->C
Msg          occured    Total        Avg         Min         Max
------------------------------------------------------------------
SQ_DESCRIBE      1    0.000720    0.000720    0.000720    0.000720
SQ_EOT           6
SQ_TUPLE        18
SQ_DONE          2
SQ_COST          2
SQ_INTERNALVER   1
SQ_PROTOCOLS     1
------------------------------------------------------------------
```

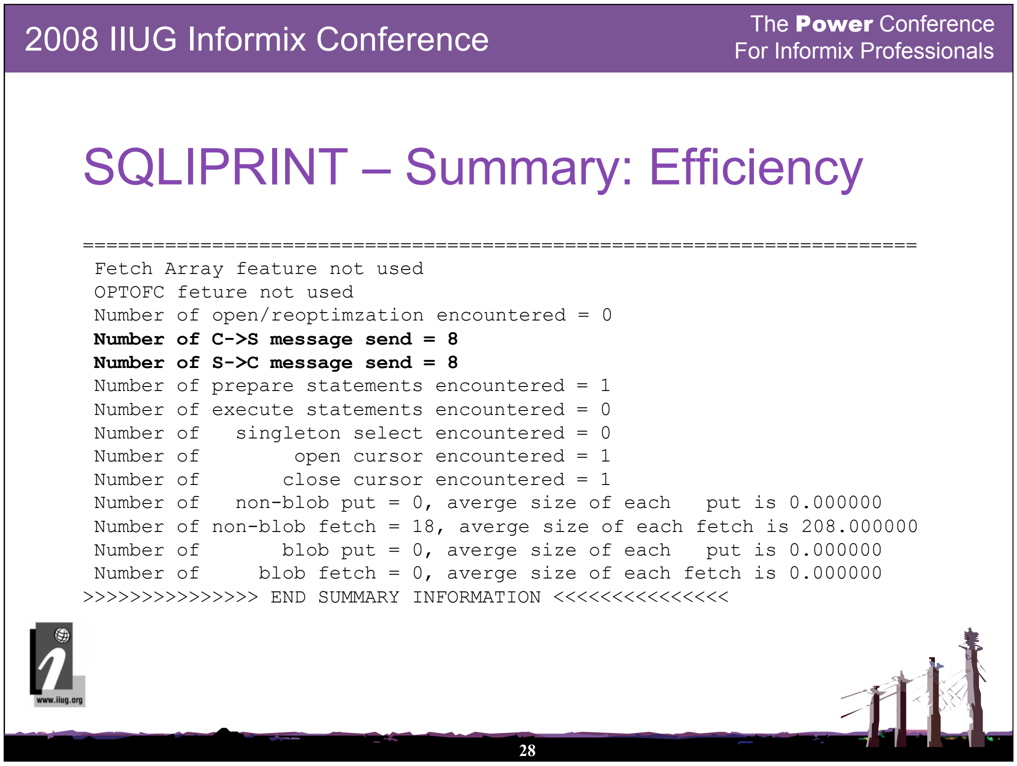
The third section is also a summary of types of messages sent, this time from the server to the client, along with some timings.



SQLIPRINT – Summary: Commands

```
======================================================================
  COMMAND TEXT INFORMATION (first 99 are listed)
a(    b) CMD[##]= '  first 60 bytes of cmd text ‘
where a = 'P'repare, or 'N'ot prepared, and b = length of command string
----------------------------------------------------------------------
P(   43) CMD[ 0]='SELECT * FROM onektup WHERE onepercent = ?;‘
----------------------------------------------------------------------
```

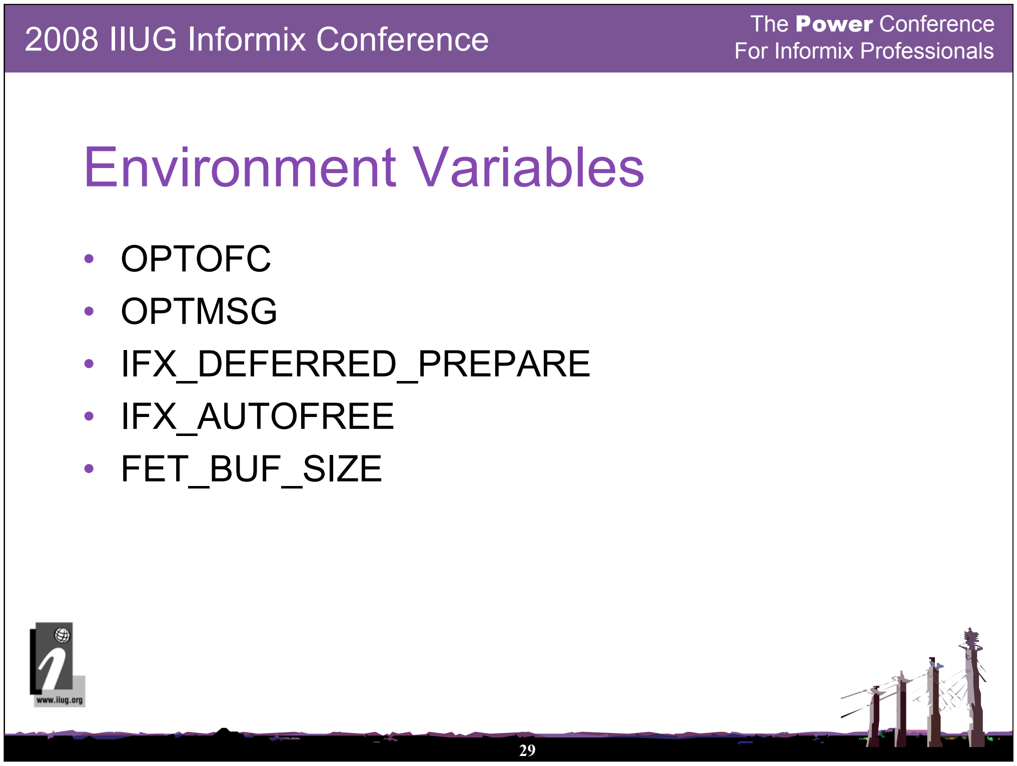
Section four is a listing of up to first 99 SQL commands sent and whether they were prepared or not.



SQLIPRINT – Summary: Efficiency

```
=======================================================================
 Fetch Array feature not used
 OPTOFC feture not used
 Number of open/reoptimzation encountered = 0
 Number of C->S message send = 8
 Number of S->C message send = 8
 Number of prepare statements encountered = 1
 Number of execute statements encountered = 0
 Number of   singleton select encountered = 0
 Number of        open cursor encountered = 1
 Number of       close cursor encountered = 1
 Number of   non-blob put = 0, averge size of each   put is 0.000000
 Number of non-blob fetch = 18, averge size of each fetch is 208.000000
 Number of       blob put = 0, averge size of each   put is 0.000000
 Number of     blob fetch = 0, averge size of each fetch is 0.000000
>>>>>>>>>>>>>>> END SUMMARY INFORMATION <<<<<<<<<<<<<<<
```

The fifth section lists, among other items, the total number of C->S messages and total number of S->C messages sent.



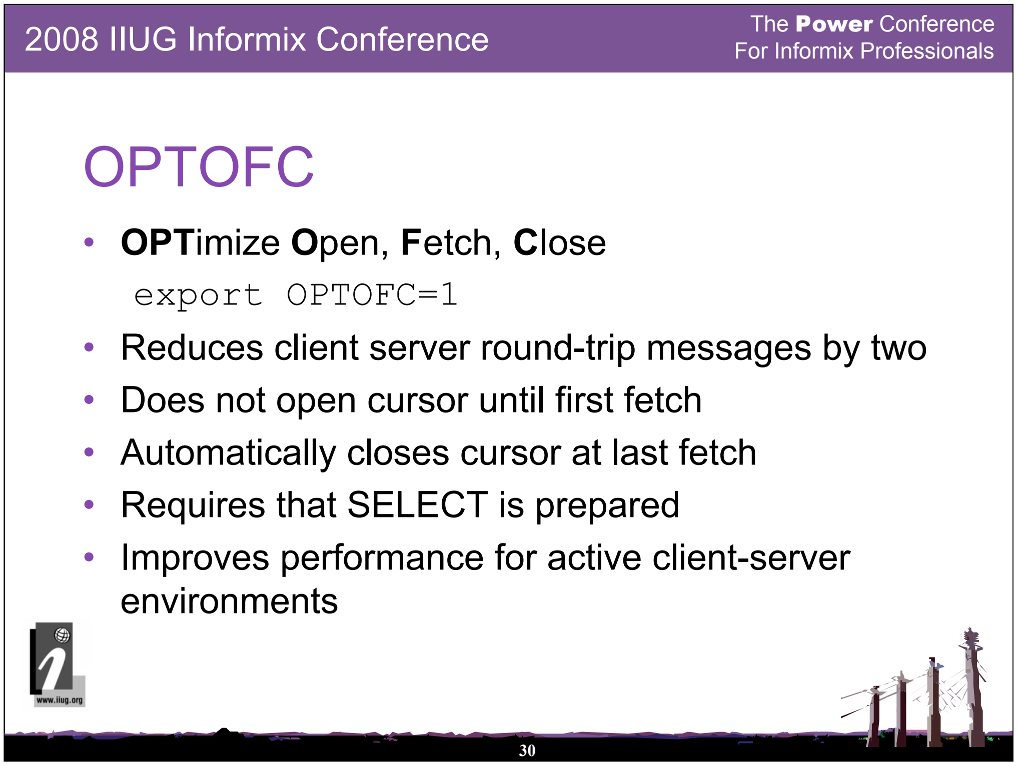
2008 IIUG Informix Conference

The **Power** Conference
For Informix Professionals

Environment Variables

- OPTOFC
- OPTMSG
- IFX_DEFERRED_PREPARE
- IFX_AUTOFREE
- FET_BUF_SIZE

There are a number of environment variables that can be used to affect the number of communications packets transmitted.



2008 IIUG Informix Conference

The **Power** Conference
For Informix Professionals

OPTOFC

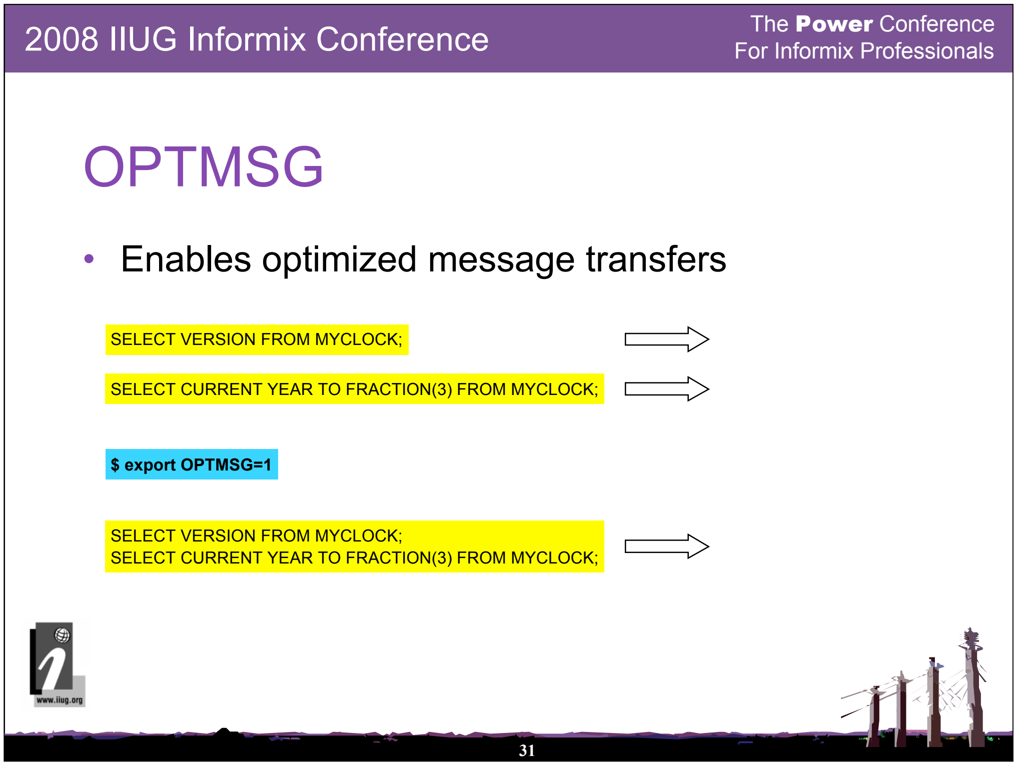
- **OPT**imize **O**pen, **F**etch, **C**lose

```
export OPTOFC=1
```

- Reduces client server round-trip messages by two
- Does not open cursor until first fetch
- Automatically closes cursor at last fetch
- Requires that SELECT is prepared
- Improves performance for active client-server environments

www.iiug.org

Setting OPTOFC will reduce the number of messages by two because the cursor is not opened until the first fetch is sent, and the cursor is automatically closed at the last fetch.



OPTMSG

- Enables optimized message transfers

```
SELECT VERSION FROM MYCLOCK;
```

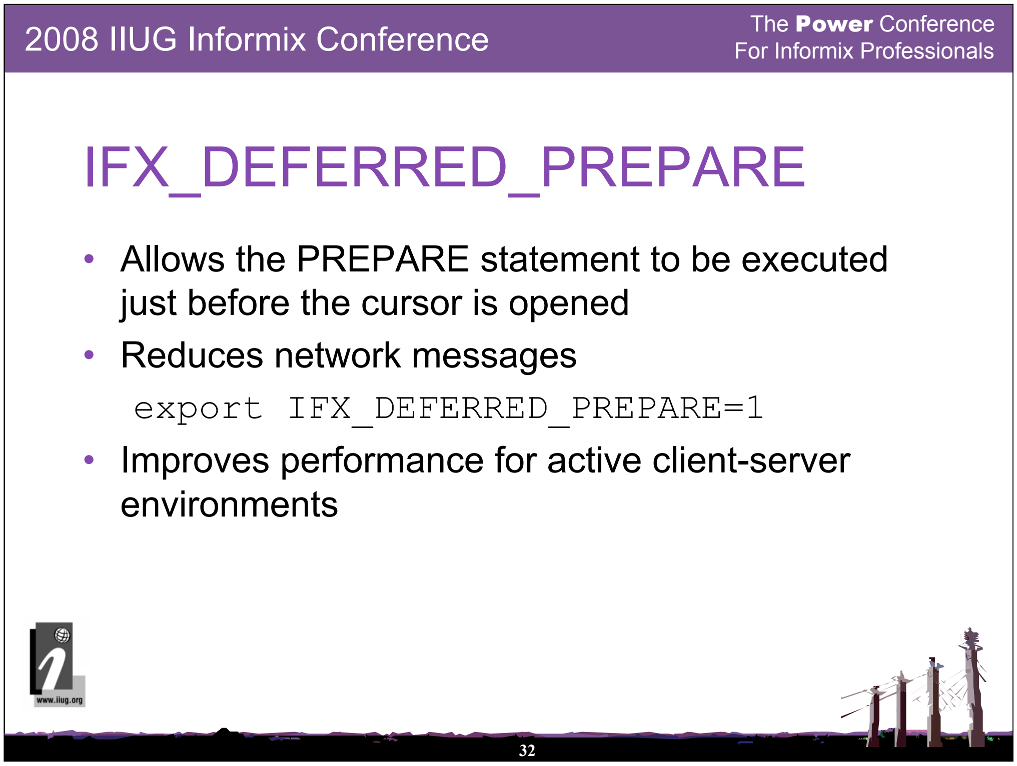
```
SELECT CURRENT YEAR TO FRACTION(3) FROM MYCLOCK;
```

```
$ export OPTMSG=1
```

```
SELECT VERSION FROM MYCLOCK;
SELECT CURRENT YEAR TO FRACTION(3) FROM MYCLOCK;
```

www.iiug.org

Setting OPTMSG will combine message traffic when possible.



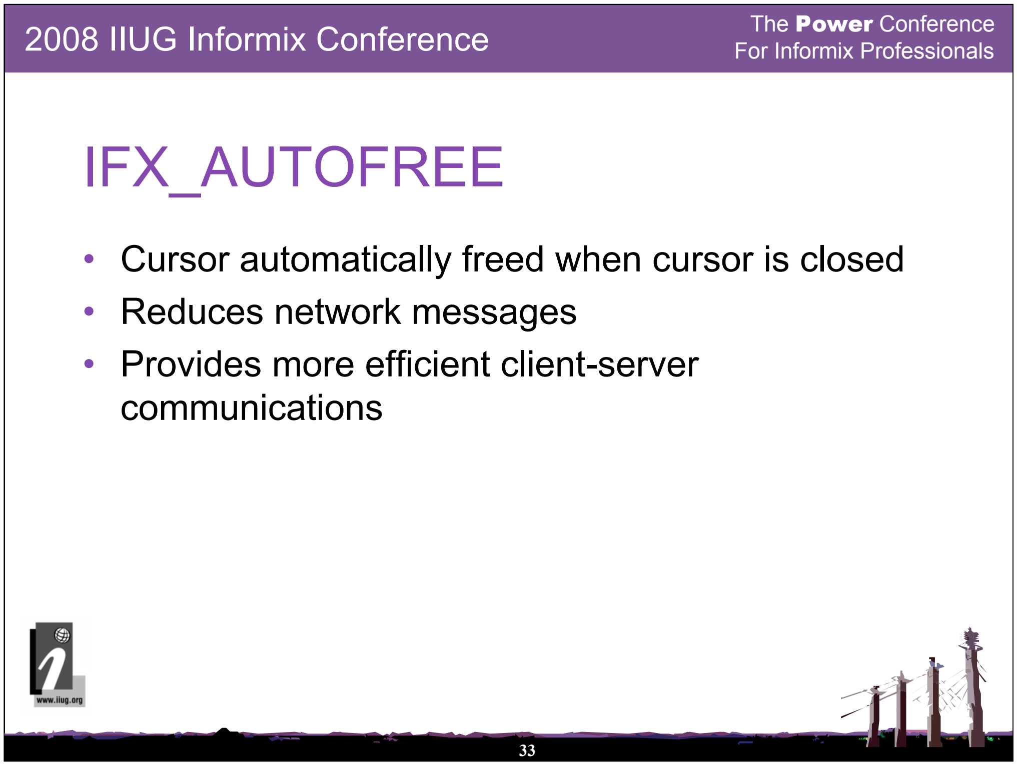
IFX_DEFERRED_PREPARE

- Allows the PREPARE statement to be executed just before the cursor is opened
- Reduces network messages

```
export IFX_DEFERRED_PREPARE=1
```

- Improves performance for active client-server environments

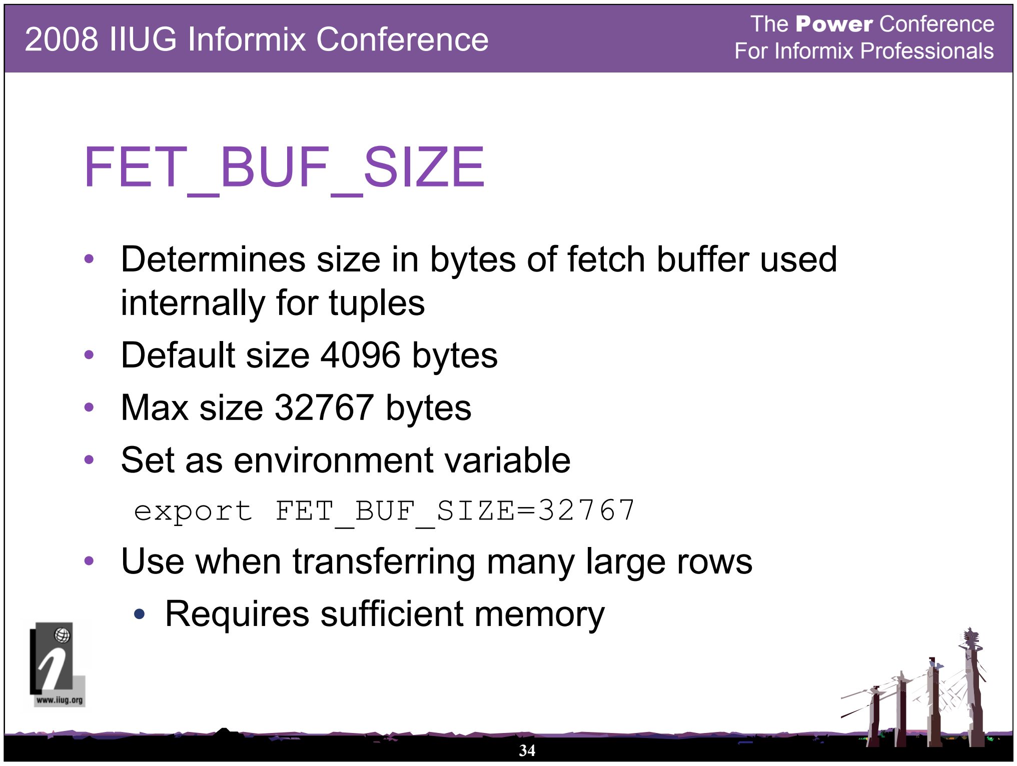
Setting IFX_DEFERRED_PREPARE will allow the prepare statement to be executed just before the cursor is opened.



IFX_AUTOFREE

- Cursor automatically freed when cursor is closed
- Reduces network messages
- Provides more efficient client-server communications

Setting IFX_AUTOFREE will automatically free the cursor when it is closed.



FET_BUF_SIZE

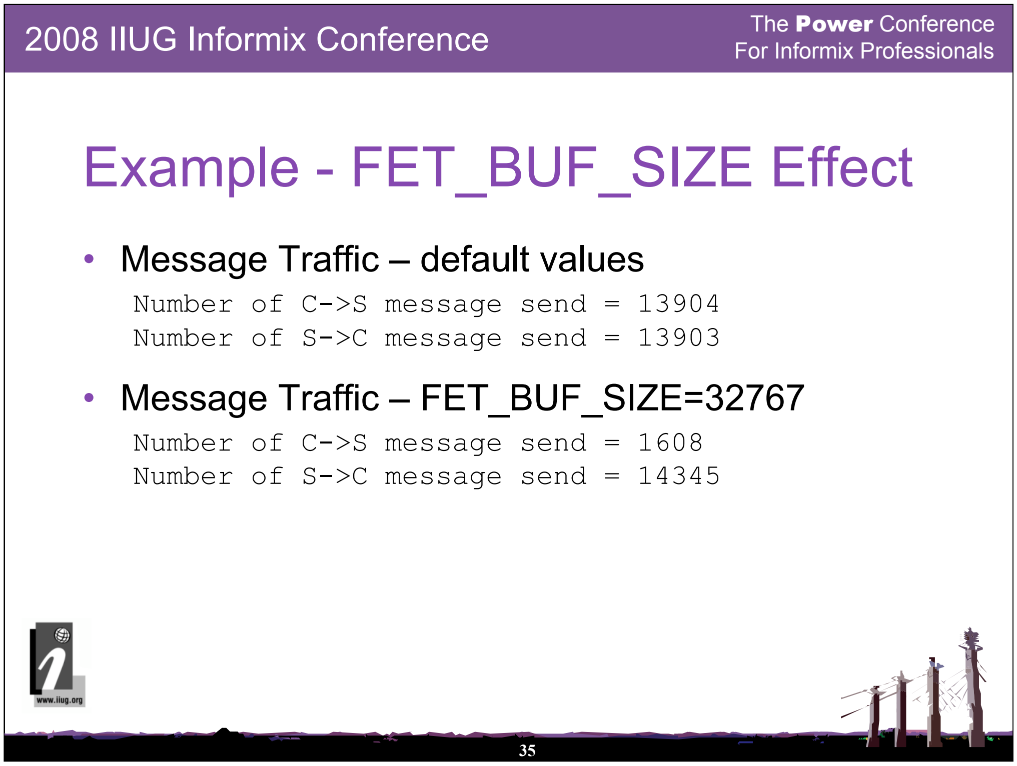
- Determines size in bytes of fetch buffer used internally for tuples
- Default size 4096 bytes
- Max size 32767 bytes
- Set as environment variable

```
export FET_BUF_SIZE=32767
```

- Use when transferring many large rows
  - Requires sufficient memory

www.iiug.org

The FET_BUF_SIZE environment variable sets the size of the client-side communications buffer.



Example - FET_BUF_SIZE Effect

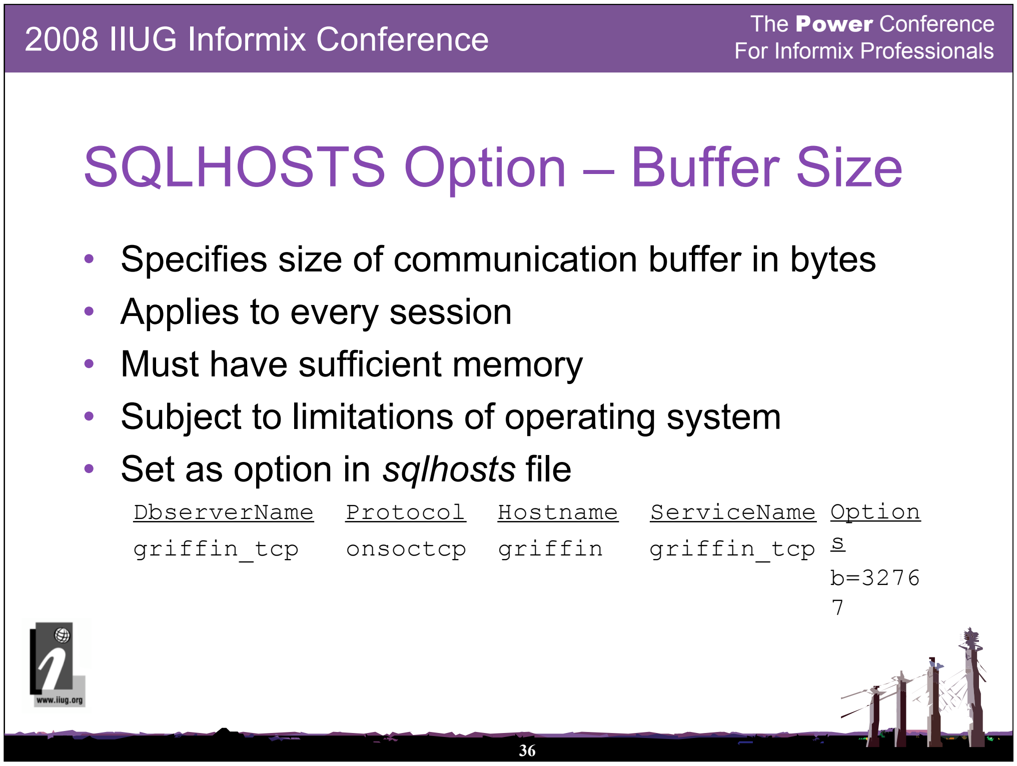
- Message Traffic – default values

```
Number of C->S message send = 13904
Number of S->C message send = 13903
```

- Message Traffic – FET_BUF_SIZE=32767

```
Number of C->S message send = 1608
Number of S->C message send = 14345
```

This slide shows the effect of FET_BUF_SIZE on client-server communications.

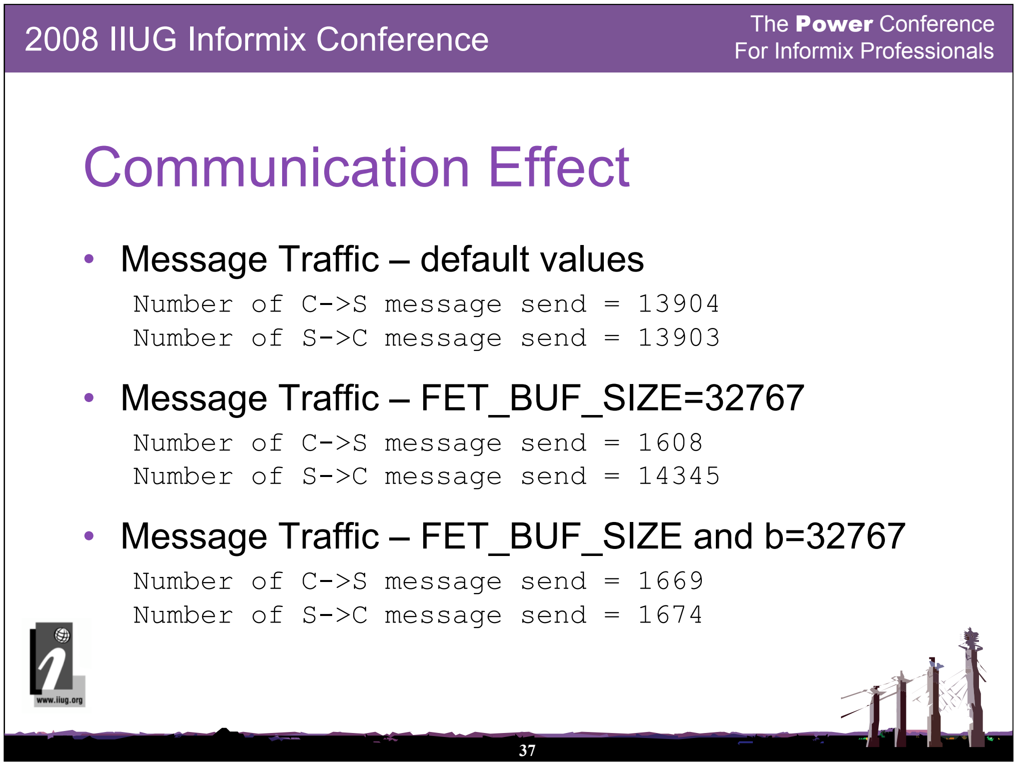


SQLHOSTS Option – Buffer Size

- Specifies size of communication buffer in bytes
- Applies to every session
- Must have sufficient memory
- Subject to limitations of operating system
- Set as option in *sqlhosts* file

DbserverName	Protocol	Hostname	ServiceName	Options
griffin_tcp	onsoctcp	griffin	griffin_tcp	b=32767

Another often overlooked possibility in tuning client-server communications is the optional buffer size parameter in the SQLHOSTS file.



Communication Effect

- Message Traffic – default values

```
Number of C->S message send = 13904
Number of S->C message send = 13903
```

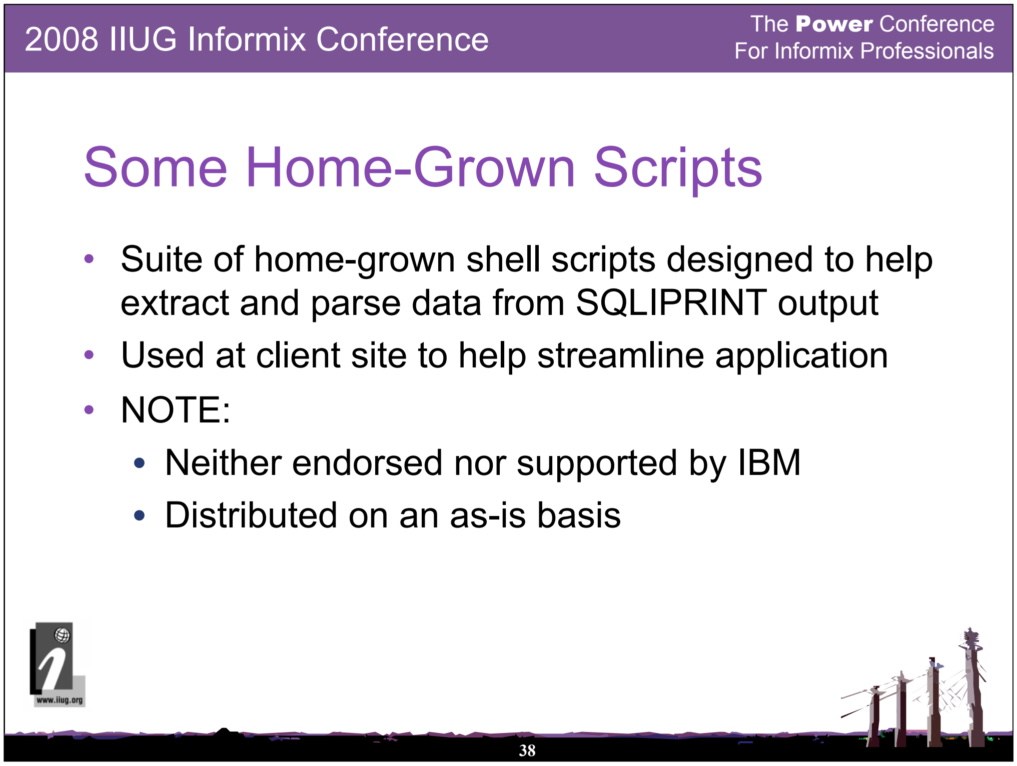
- Message Traffic – FET_BUF_SIZE=32767

```
Number of C->S message send = 1608
Number of S->C message send = 14345
```

- Message Traffic – FET_BUF_SIZE and b=32767

```
Number of C->S message send = 1669
Number of S->C message send = 1674
```

This slide shows the effect of the size of the communications buffer on message traffic.

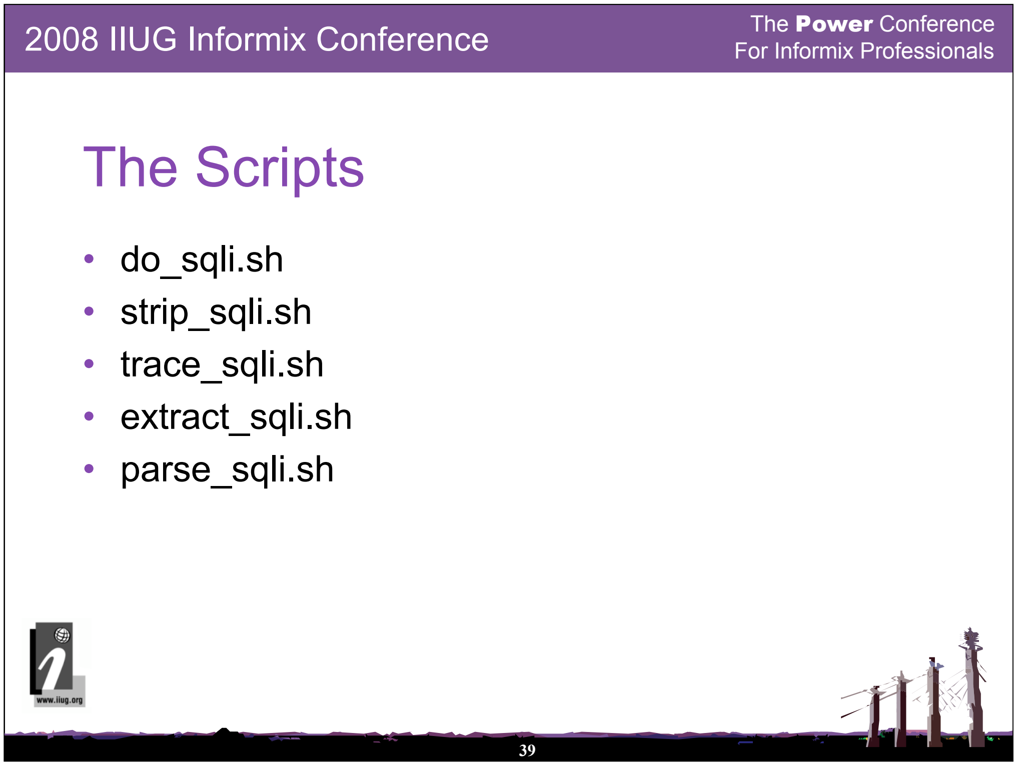


Some Home-Grown Scripts

- Suite of home-grown shell scripts designed to help extract and parse data from SQLIPRINT output
- Used at client site to help streamline application
- NOTE:
  - Neither endorsed nor supported by IBM
  - Distributed on an as-is basis

www.iiug.org

In this part of the session we’ll talk about a suite of scripts that were written to better understand the information produced by sqliprint.



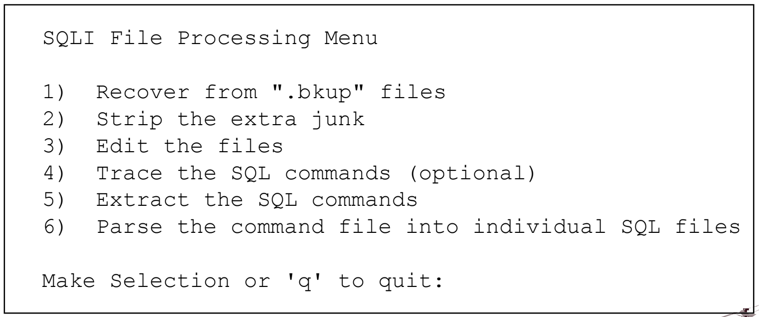
The Scripts

- do_sqli.sh
- strip_sqli.sh
- trace_sqli.sh
- extract_sqli.sh
- parse_sqli.sh

This is a list of the shell scripts in the suite.

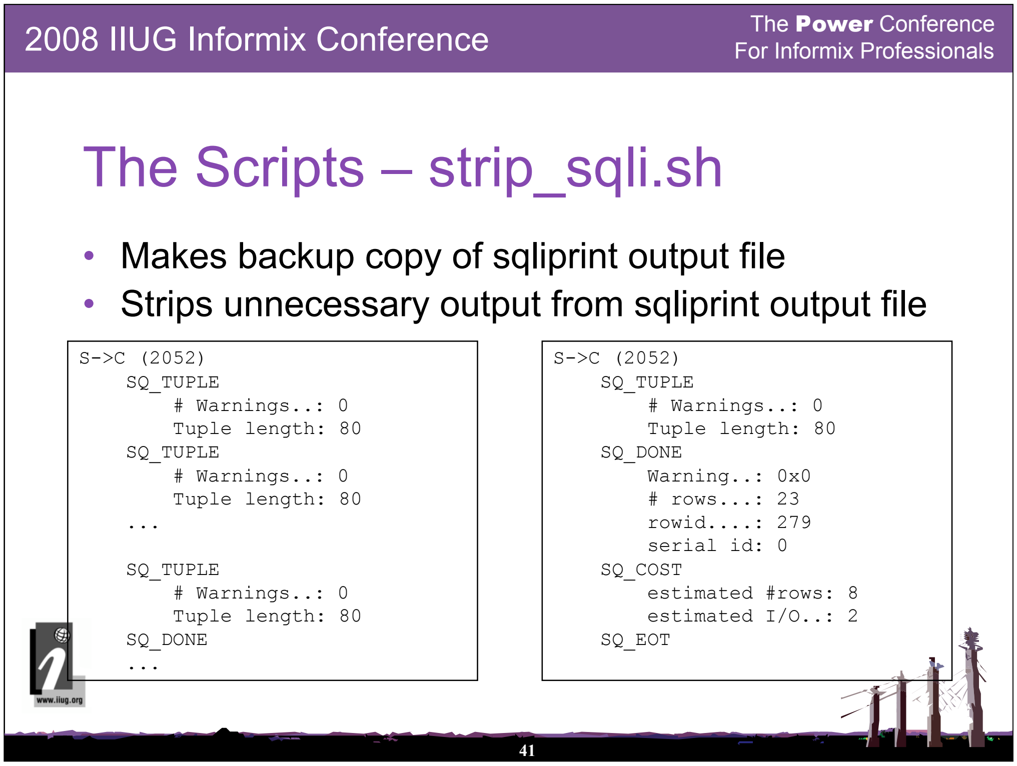
The Scripts – do_sqli.sh

- A menu that drives other scripts
- Requires file named **sqlifiles**
  - Contains list of output files from sqliprint



```
SQLI File Processing Menu

1)   Recover from ".bkup" files
2)   Strip the extra junk
3)   Edit the files
4)   Trace the SQL commands (optional)
5)   Extract the SQL commands
6)   Parse the command file into individual SQL files

Make Selection or 'q' to quit:
```

The do_sqli.sh script is a menu that drives the rest of the process (shown above).



The Scripts – strip_sqli.sh

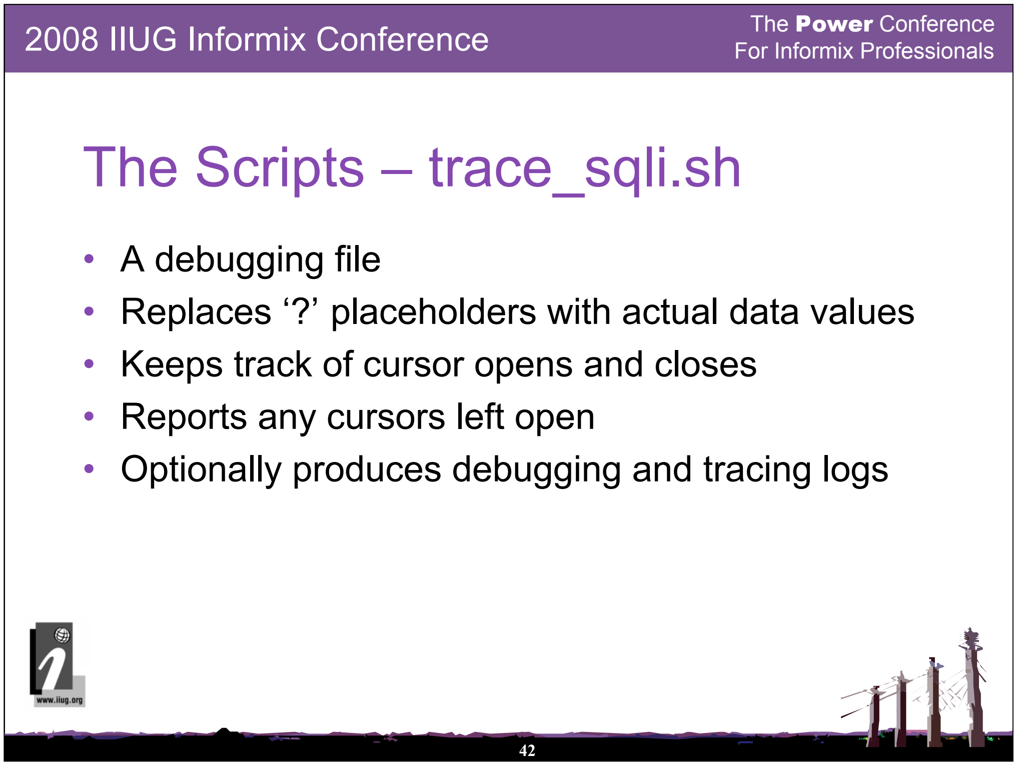
- Makes backup copy of sqliprint output file
- Strips unnecessary output from sqliprint output file

```
S->C (2052)
    SQ_TUPLE
        # Warnings..: 0
        Tuple length: 80
    SQ_TUPLE
        # Warnings..: 0
        Tuple length: 80
    ...

    SQ_TUPLE
        # Warnings..: 0
        Tuple length: 80
    SQ_DONE
    ...
```

```
S->C (2052)
    SQ_TUPLE
        # Warnings..: 0
        Tuple length: 80
    SQ_DONE
        Warning..: 0x0
        # rows...: 23
        rowid....: 279
        serial id: 0
    SQ_COST
        estimated #rows: 8
        estimated I/O..: 2
    SQ_EOT
```

The first thing strip_sqli.sh (menu step 2) does is make a backup copy of the sqliprint output file, giving it a “.bkup” file extension.

Next, as we can see in the code samples, it removes repeating SQ_TUPLE groups to shorten the file and make it more manageable.

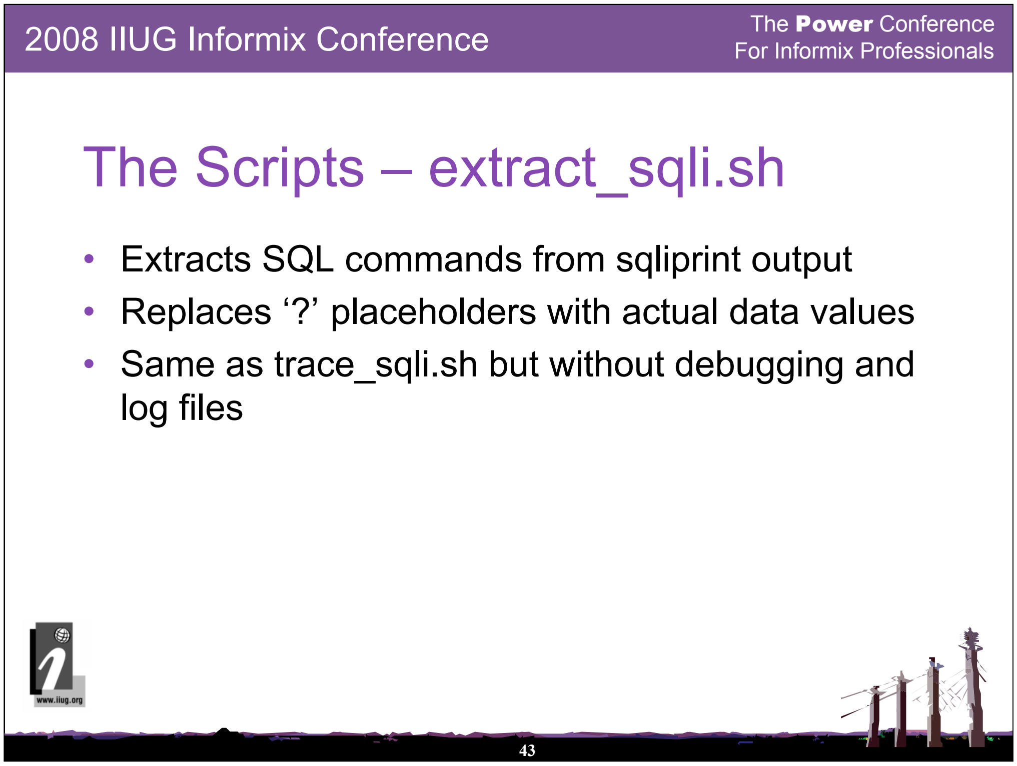


The Scripts – trace_sqli.sh

- A debugging file
- Replaces '?' placeholders with actual data values
- Keeps track of cursor opens and closes
- Reports any cursors left open
- Optionally produces debugging and tracing logs

The trace_sqli.sh (menu step 4) script lists the SQL commands, their corresponding server statement IDs, and substitutes actual data values for the '?' placeholders.

It also keeps track of the number of cursors opened and closed and reports on the number left open.

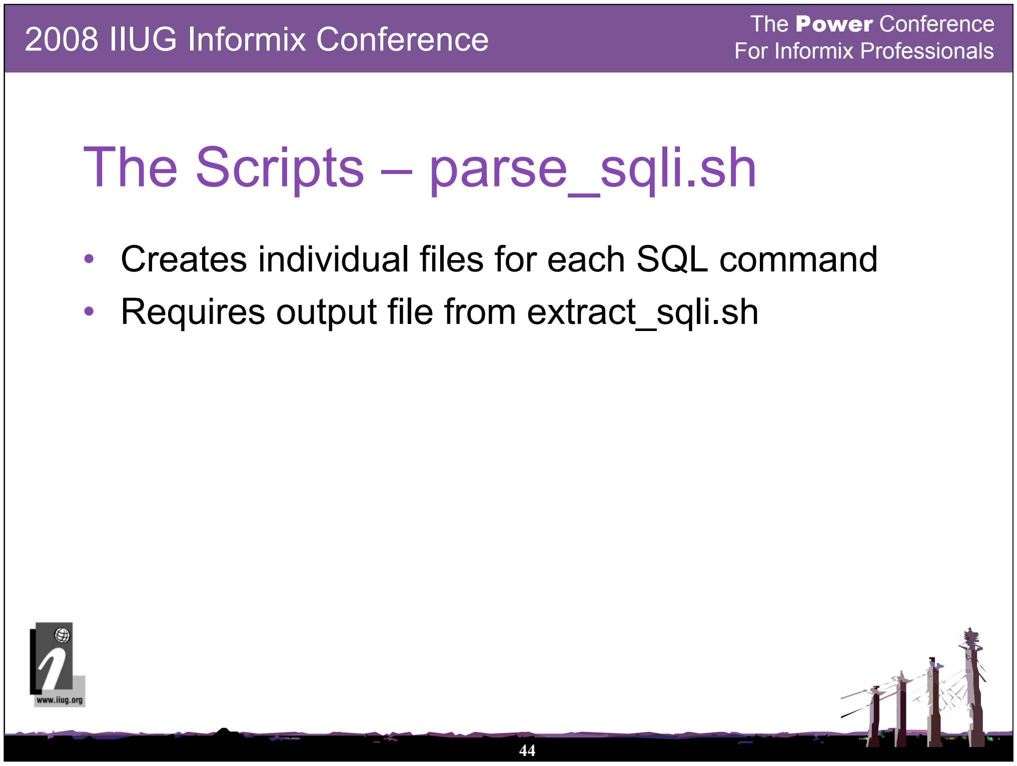


The Scripts – extract_sqli.sh

- Extracts SQL commands from sqliprint output
- Replaces ‘?’ placeholders with actual data values
- Same as trace_sqli.sh but without debugging and log files

The extract_sqli.sh (menu step 5) script is the same as the trace_sqli.sh (menu step 4) script, but without the TRACE and DEBUG options.

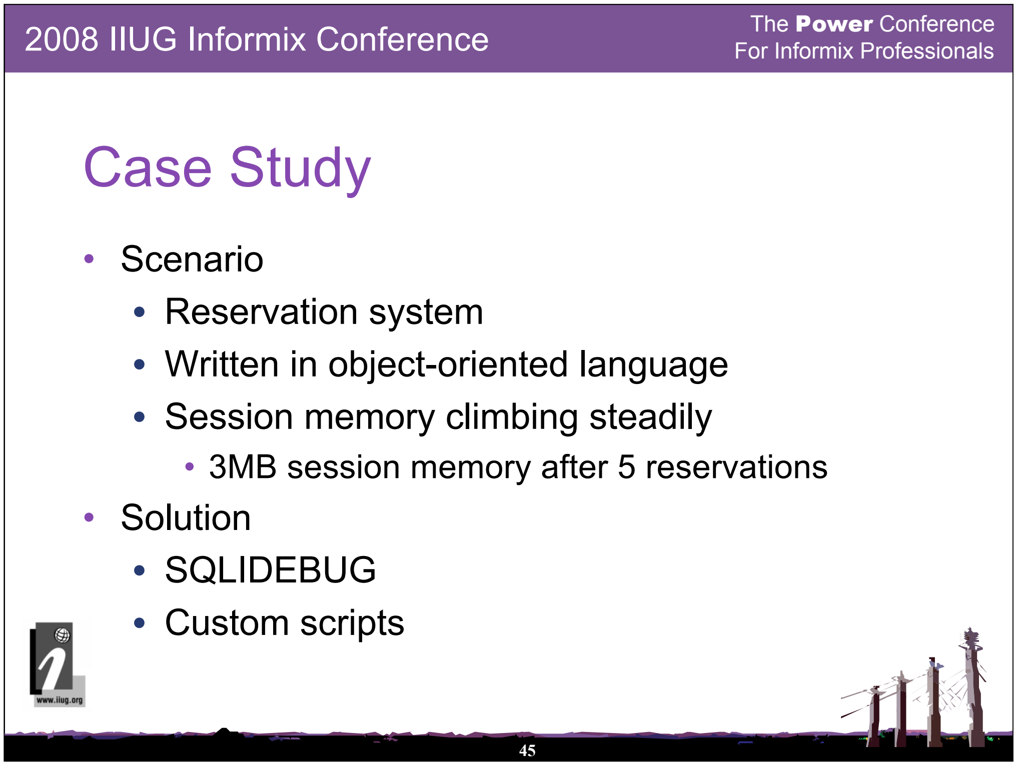
It produces an output file named <file>.extract, which is a list of the SQL commands with actual data values substituted for the ‘?’ placeholders.



The Scripts – parse_sqli.sh

- Creates individual files for each SQL command
- Requires output file from extract_sqli.sh

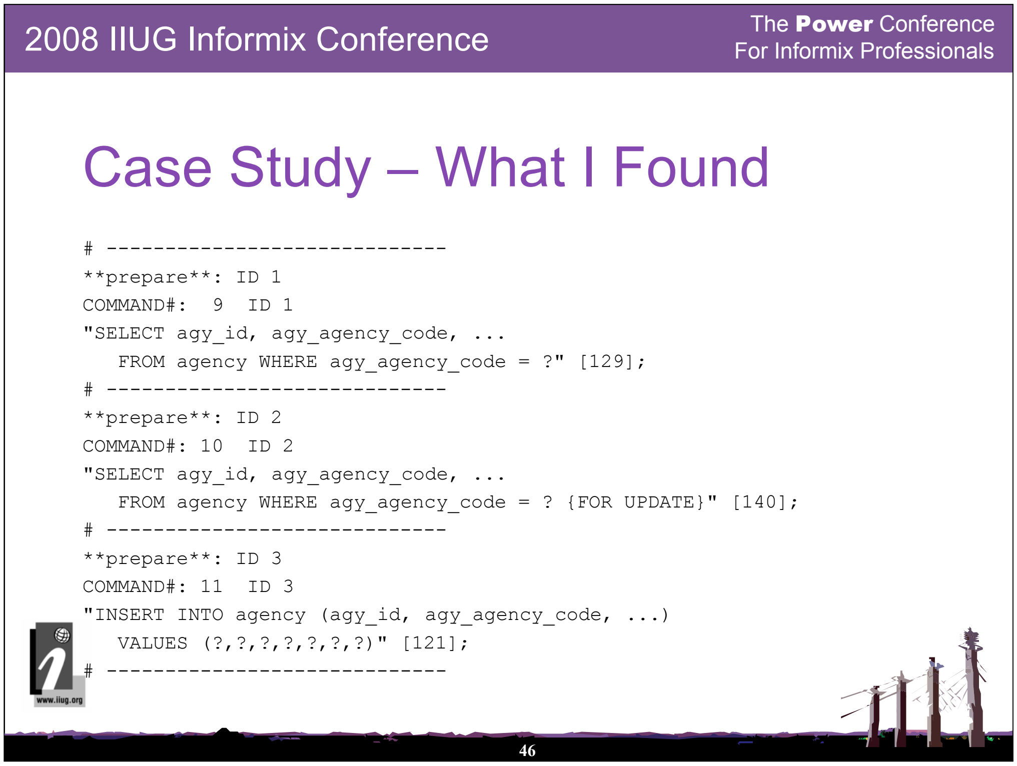
The parse_sqli.sh (menu step 6) script takes the SQL commands listed by the extract_sqli.sh (menu step 5) process and places each command into a separate SQL file.



Case Study

- Scenario
  - Reservation system
  - Written in object-oriented language
  - Session memory climbing steadily
    - 3MB session memory after 5 reservations
- Solution
  - SQLIDEBUG
  - Custom scripts

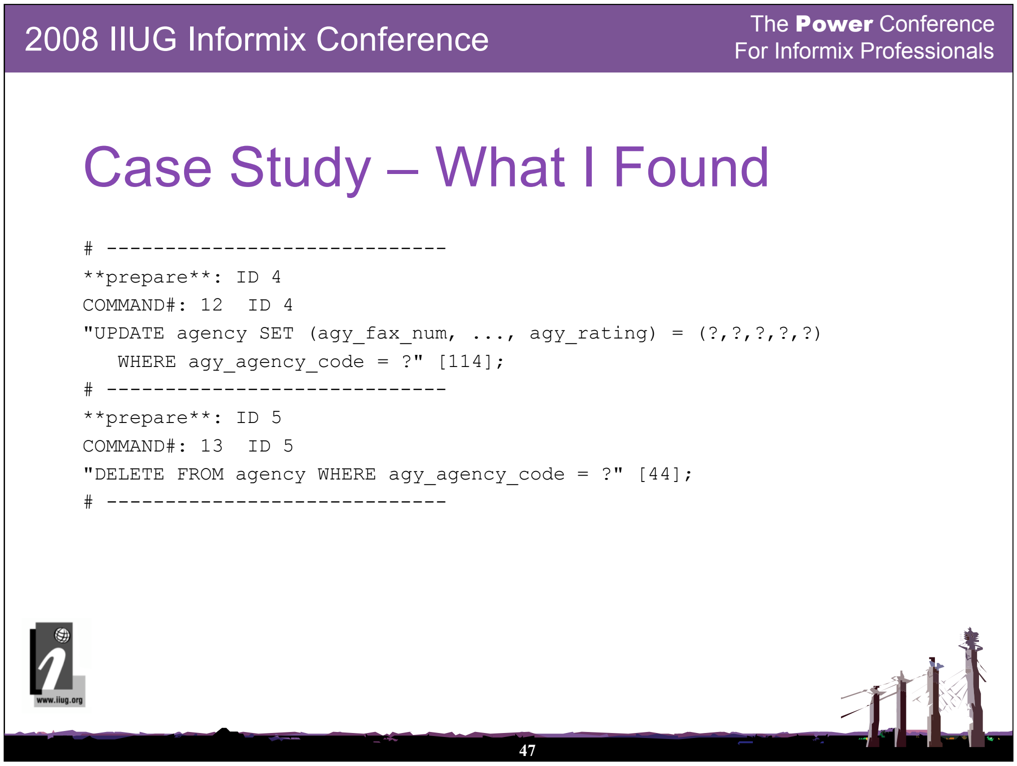
I wrote these scripts because I was working on a project where the application developers were writing a reservation system.



Case Study – What I Found

```
# -----------------------------
**prepare**: ID 1
COMMAND#:  9  ID 1
"SELECT agy_id, agy_agency_code, ...
   FROM agency WHERE agy_agency_code = ?" [129];
# -----------------------------
**prepare**: ID 2
COMMAND#: 10  ID 2
"SELECT agy_id, agy_agency_code, ...
   FROM agency WHERE agy_agency_code = ? {FOR UPDATE}" [140];
# -----------------------------
**prepare**: ID 3
COMMAND#: 11  ID 3
"INSERT INTO agency (agy_id, agy_agency_code, ...)
   VALUES (?,?,?,?,?,?,?)" [121];
# -----------------------------
```

The first of these was revealed by the output of the scripts, and the slide shows how I was able to identify this problem.

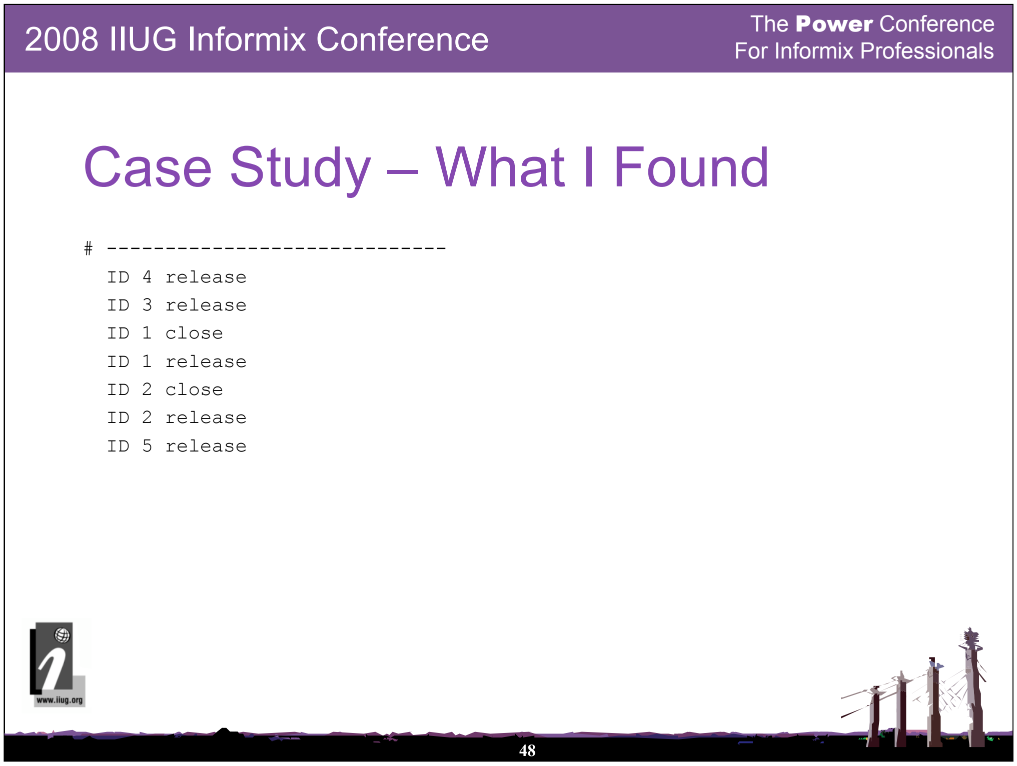


Case Study – What I Found

```
# -----------------------------
**prepare**: ID 4
COMMAND#: 12  ID 4
"UPDATE agency SET (agy_fax_num, ..., agy_rating) = (?,?,?,?,?)
   WHERE agy_agency_code = ?" [114];
# -----------------------------
**prepare**: ID 5
COMMAND#: 13  ID 5
"DELETE FROM agency WHERE agy_agency_code = ?" [44];
# -----------------------------
```

www.iiug.org

We also saw UPDATE and DELETE statements being prepared against the agency table.

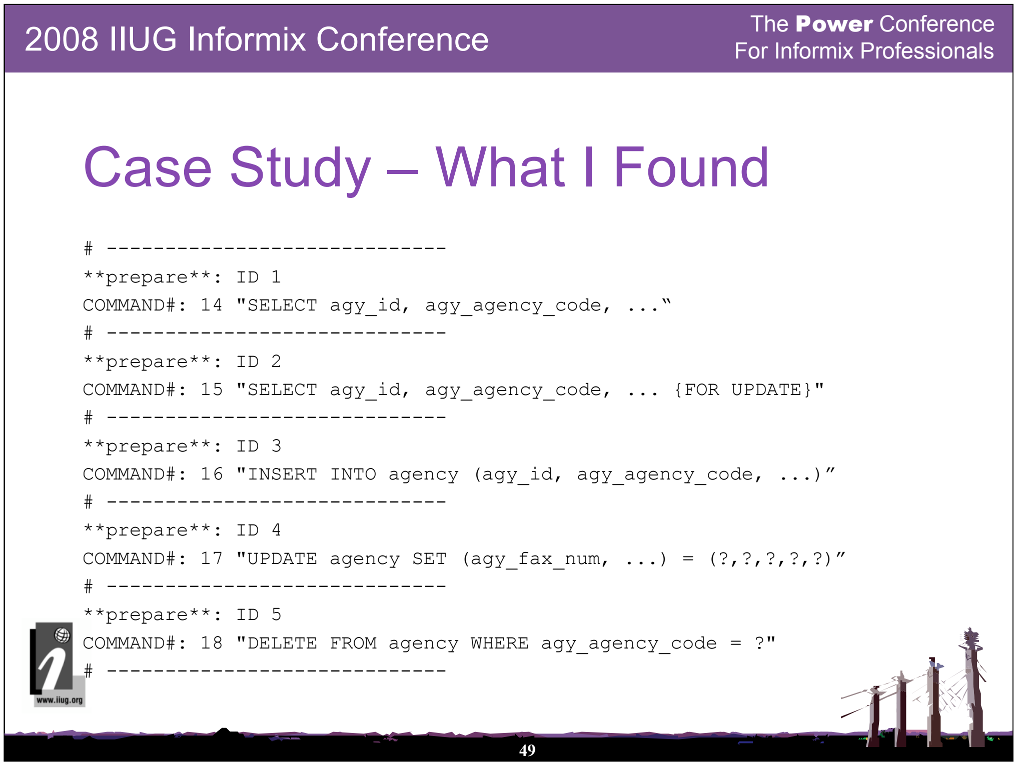


Case Study – What I Found

```
# -----------------------------
  ID 4 release
  ID 3 release
  ID 1 close
  ID 1 release
  ID 2 close
  ID 2 release
  ID 5 release
```

www.iiug.org

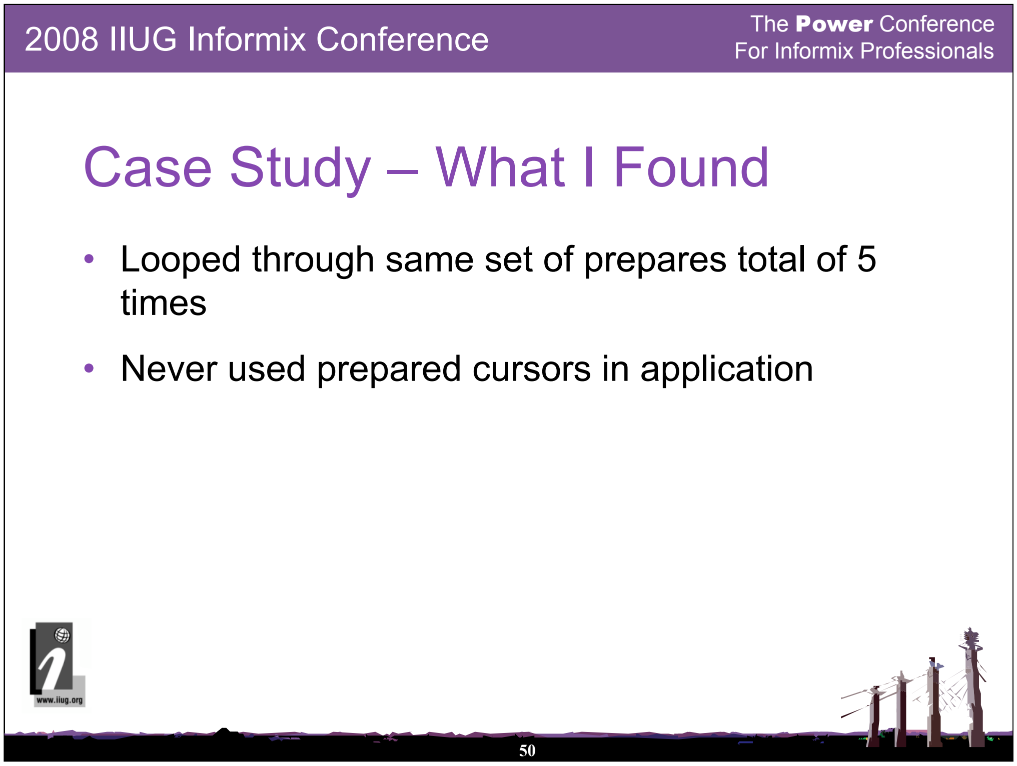
Right after the 5 statements were prepared, they were closed and released.



Case Study – What I Found

```
# -----------------------------
**prepare**: ID 1
COMMAND#: 14 "SELECT agy_id, agy_agency_code, ...“
# -----------------------------
**prepare**: ID 2
COMMAND#: 15 "SELECT agy_id, agy_agency_code, ... {FOR UPDATE}"
# -----------------------------
**prepare**: ID 3
COMMAND#: 16 "INSERT INTO agency (agy_id, agy_agency_code, ...)”
# -----------------------------
**prepare**: ID 4
COMMAND#: 17 "UPDATE agency SET (agy_fax_num, ...) = (?,?,?,?,?)”
# -----------------------------
**prepare**: ID 5
COMMAND#: 18 "DELETE FROM agency WHERE agy_agency_code = ?"
# -----------------------------
```

Immediately following the release, I found another set of prepare statements.

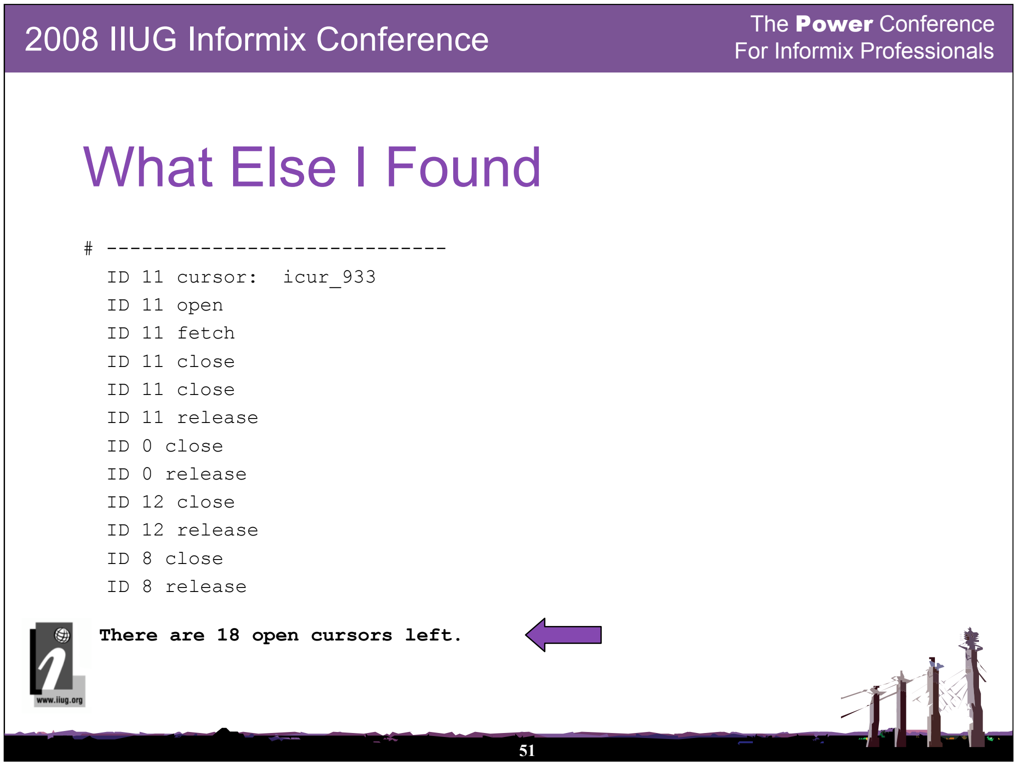


Case Study – What I Found

- Looped through same set of prepares total of 5 times
- Never used prepared cursors in application

www.iiug.org

In fact, this 5-statement set of prepares was done 5 times.

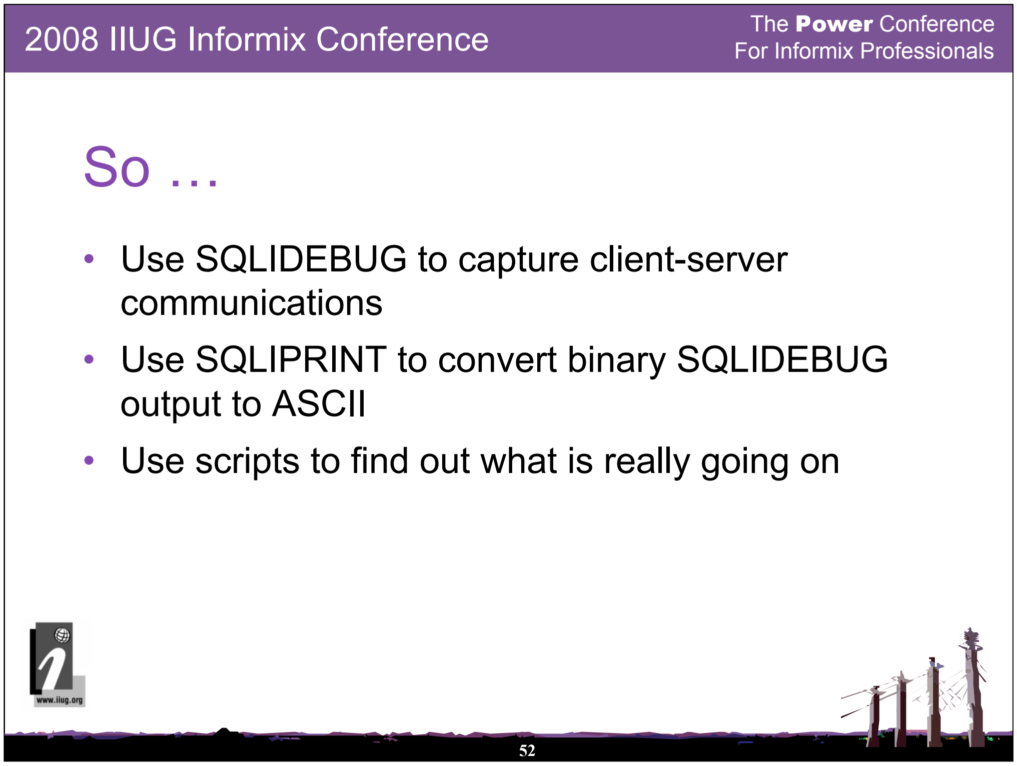


What Else I Found

```
# -----------------------------
  ID 11 cursor:  icur_933
  ID 11 open
  ID 11 fetch
  ID 11 close
  ID 11 close
  ID 11 release
  ID 0 close
  ID 0 release
  ID 12 close
  ID 12 release
  ID 8 close
  ID 8 release
```

There are 18 open cursors left. ⬅

What I also found was that at the end of each reservation cycle, 18 cursors were left open.



2008 IIUG Informix Conference

The **Power** Conference
For Informix Professionals

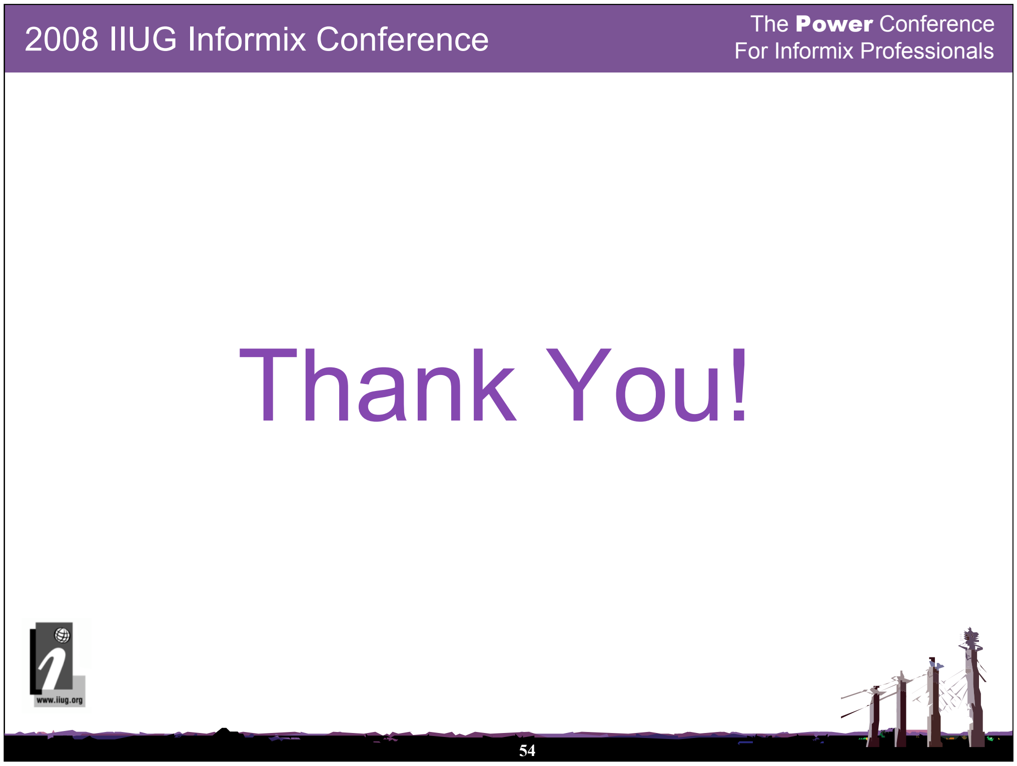
So …

- Use SQLIDEBUG to capture client-server communications
- Use SQLIPRINT to convert binary SQLIDEBUG output to ASCII
- Use scripts to find out what is really going on

www.iiug.org

So what we find is that we can use the SQLIDEBUG and sqliprint process, in conjunction with a fairly simple suite of scripts, to help us streamline our application by eliminating unnecessary communication, and even identifying certain types of problems with program logic.

Questions?



Thank You!

Session D15

Using SQLIDEBUG to Help Streamline Your Application

Mike Lowe

IBM

mike.lowe@us.ibm.com