Best Practices for Developing High Performing Java Applications with Db2

Maryela Weihrauch
IBM Distinguished Engineer, WW Data & AI on System z
weihrau@us.ibm.com

Acknowledgements and Disclaimers:

Availability. References in this presentation to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates.

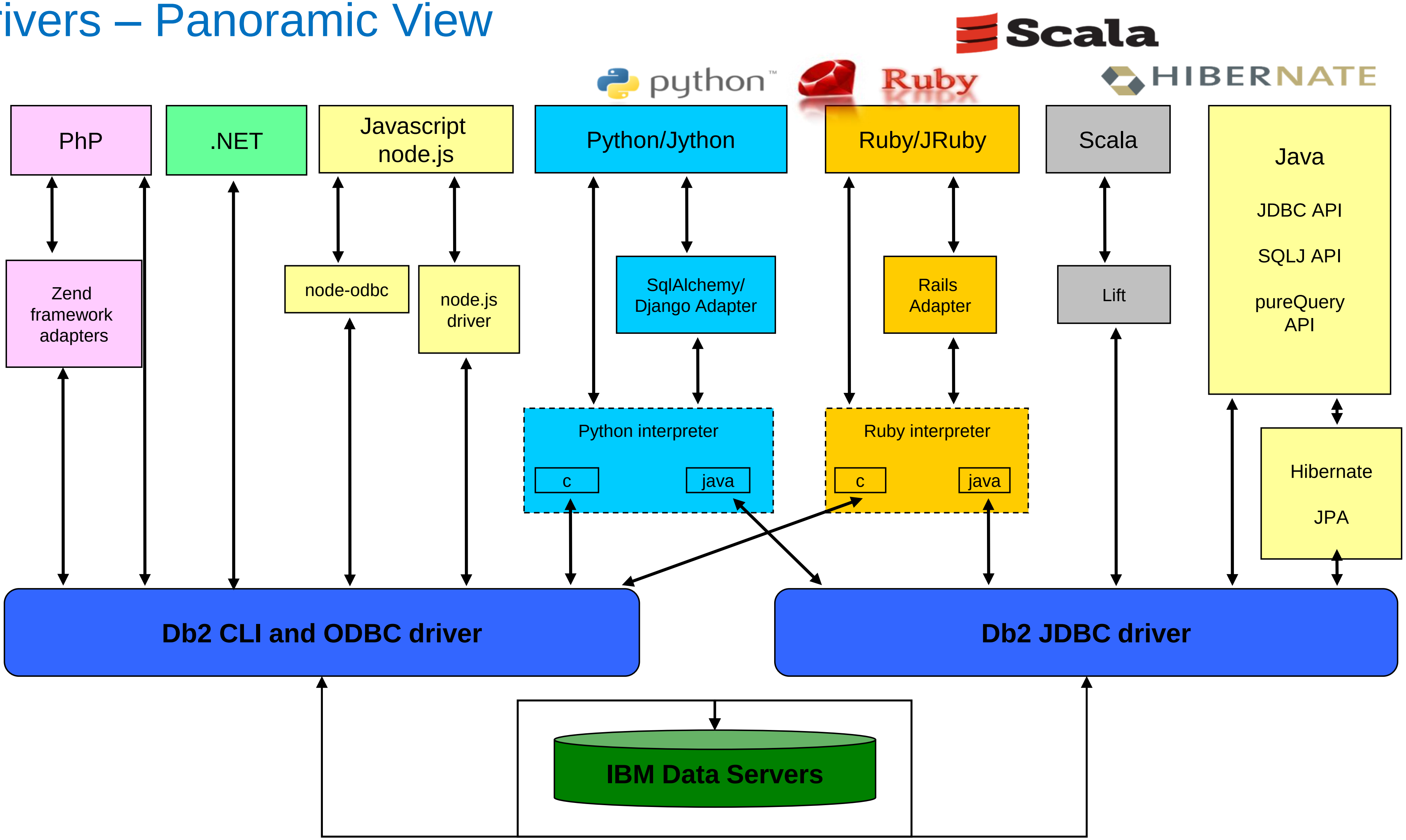
The workshops, sessions and materials have been prepared by IBM or the session speakers and reflect their own views. They are provided for informational purposes only, and are neither intended to, nor shall have the effect of being, legal or other guidance or advice to any participant. While efforts were made to verify the completeness and accuracy of the information contained in this presentation, it is provided AS-IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, this presentation or any other materials. Nothing contained in this presentation is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer. Nothing contained in these materials is intended to, nor shall have the effect of, stating or implying that any activities undertaken by you will result in any specific sales, revenue growth or other results.

Agenda

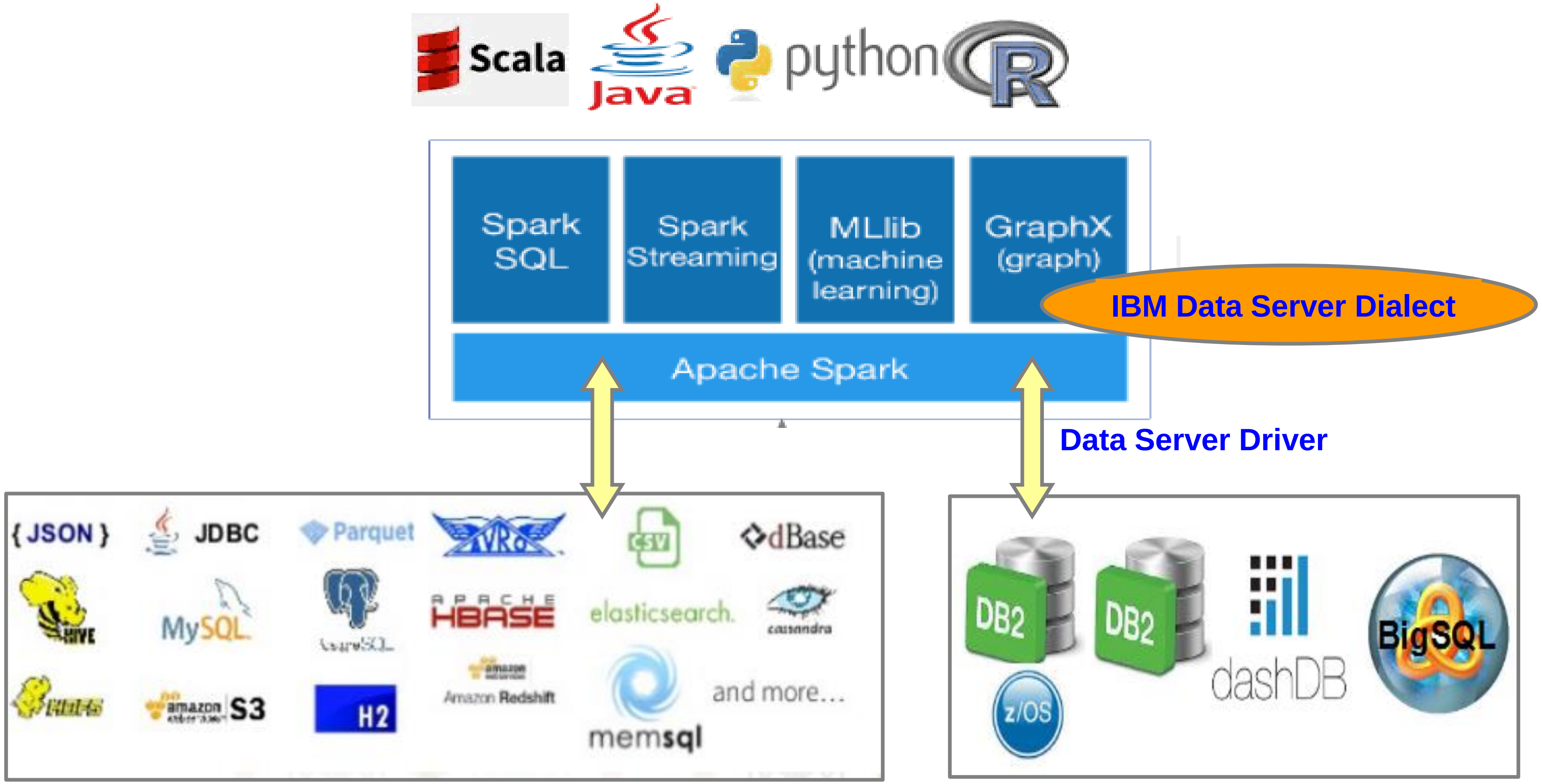
- Db2 JDBC driver architecture and API details
- Connection Management
- Db2 Dynamic Statement Cache
- Best practice for SQL Execution

Db2 Drivers – Panoramic View

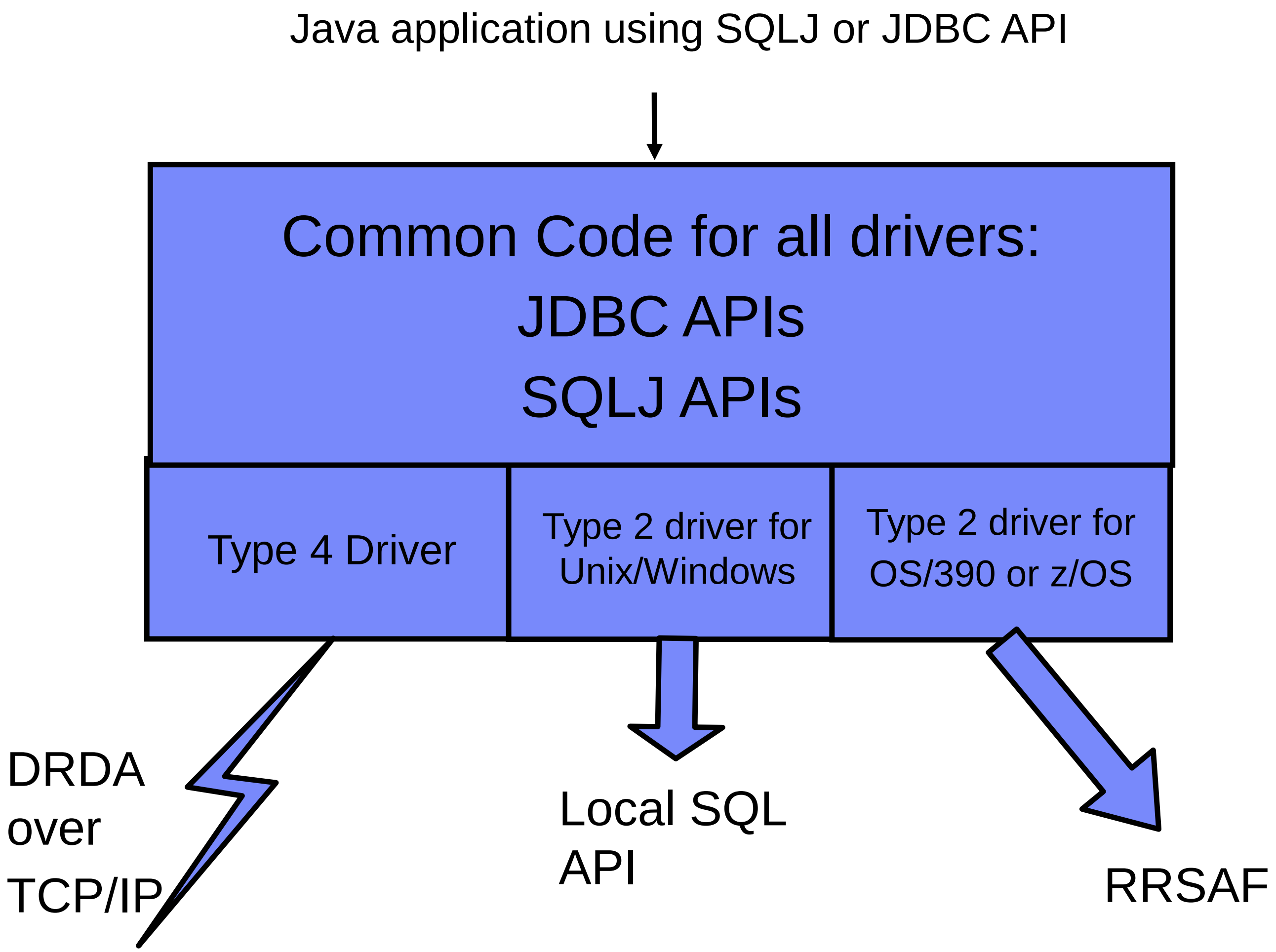


Integration with Spark

- Goal – Seamless read from IBM data servers into Spark & write from Spark DataFrames to IBM data servers



Data Server Driver for JDBC/SQLJ - Architecture



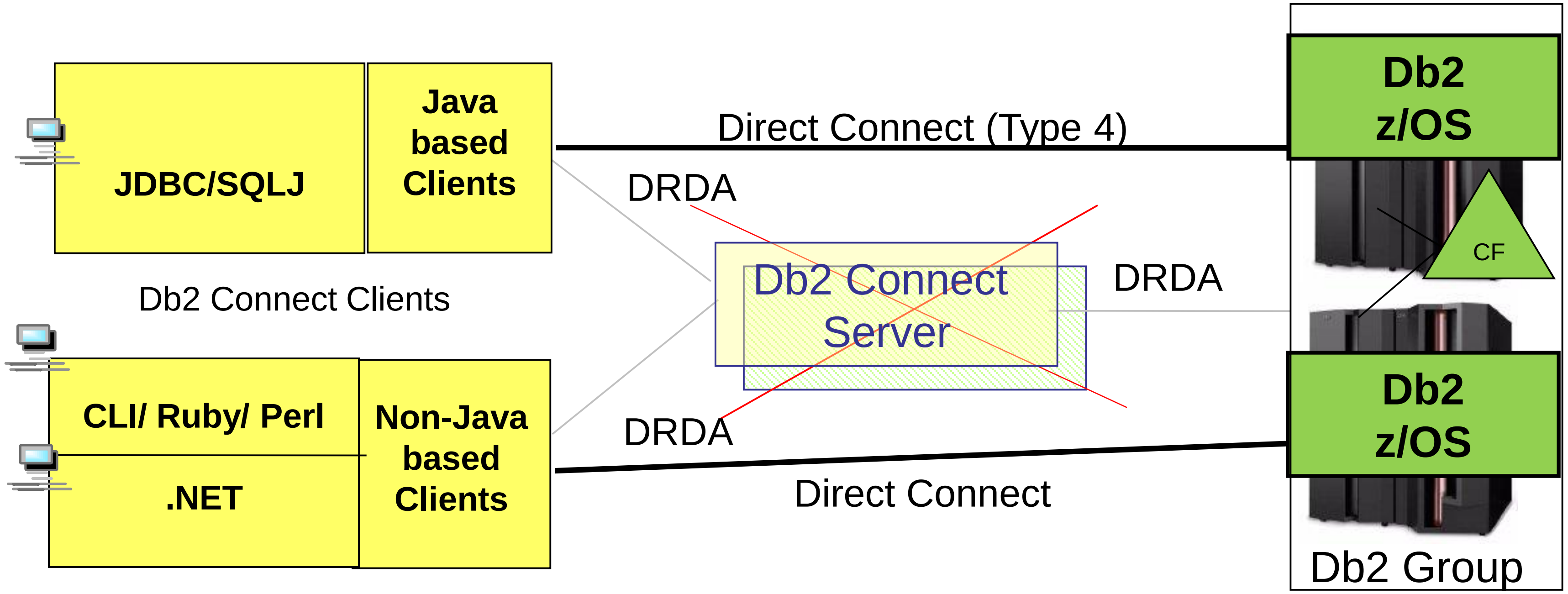
Features

- Full function JDBC driver, much more than a JDBC driver

Feature	Details
High Availability	Work load Balancing, Client re-route (Failover)
Scalability	Extensive Multi-threading; Better resource utilization
Performance	Connection Reuse/ efficient socket management; DRDA chaining / batching; Defer prepares ; Statement Caching
Security	Encryption / Authentication using SSL, Kerberos
Monitoring	Extended diagnostics for system monitoring
Robust Application functionality	Internationalization, Bi-di support, Distributed transactions, JSON, Oracle Compatibility
Data types	BIGINT, TIMESTAMP, XML/BLOB/CLOB
Standards	JDBC, ODBC, XA, DRDA
Programmer Productivity	Integration with Open Source ORM frameworks

Client Configurations – Distributed Connectivity

- Based on DRDA - Distributed Relational Database Architecture



- Data Server Clients no longer require Db2 Connect Server
- Data Server Clients are Sysplex enabled with improved availability

Getting a Connection

- java.sql.DriverManager API
  - The actual driver type determined during runtime from the connection URL format:
    - Type 2 - “jdbc:db2:database”
    - Type 4 - “jdbc:db2://host:port/database” (default port number 446)

```
Class.forName(“com.ibm.db2.jcc.DB2Driver”);
 Connection con  = DriverManager.getConnection(“jdbc:db2://localhost:50000/sample”,
“username”, “password”);
```

  - Disadvantage – Reduces portability due to class name and URL.

- javax.sql.DataSource API
  - Logical name mapped to DataSource object via JNDI naming service
  - logical name -> driver info, DB name, IP, port, user, password, etc.
  - App servers used to manage & configure data sources
  - Makes application portable

```
Context ctx=new InitialContext();
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
Connection con=ds.getConnection();
```

Driver and Connection Properties

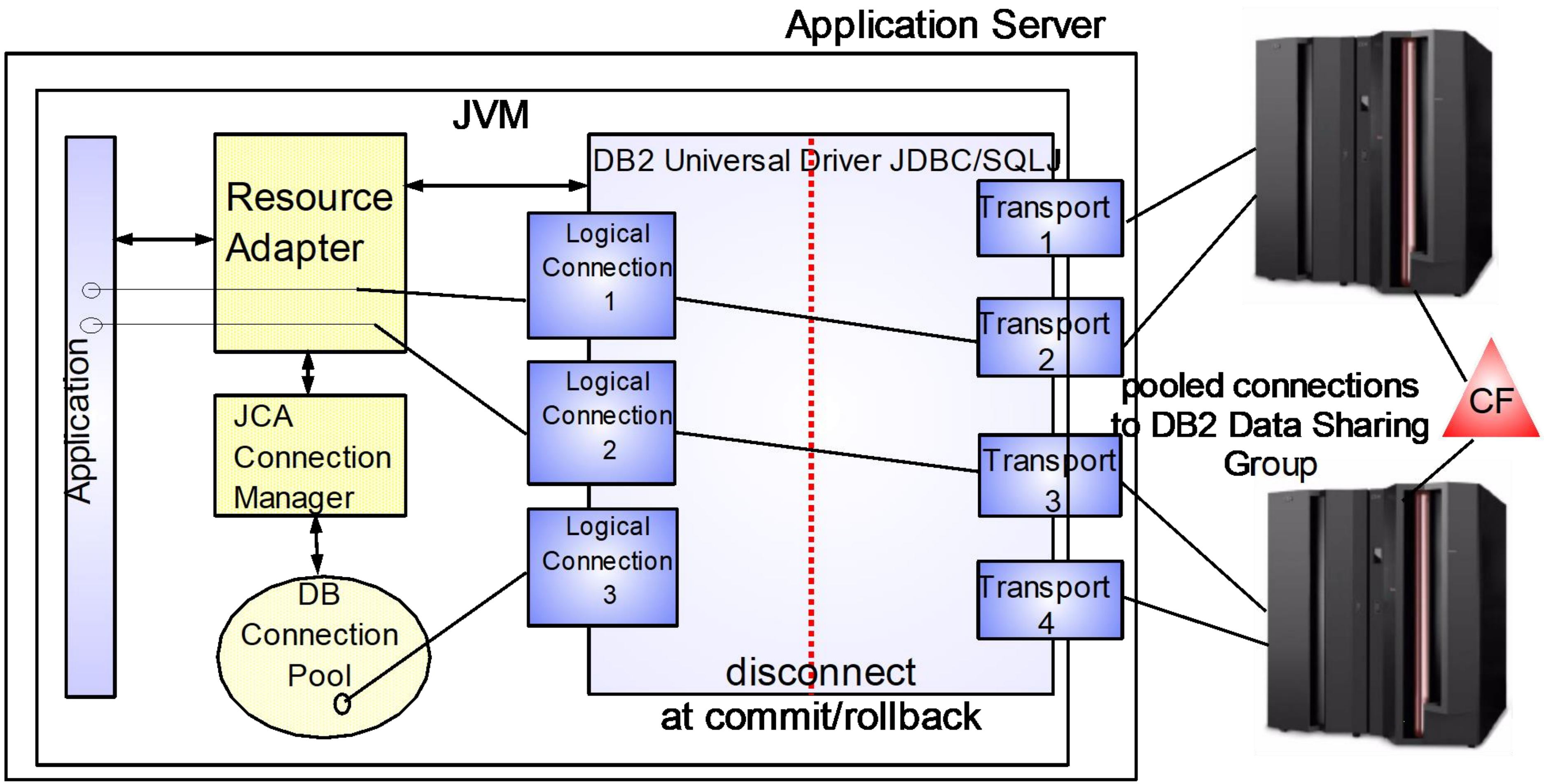
- Global driver properties can be provided through Java system properties or a properties file.
- JDBC API defines a set of properties to identify and describe a DataSource implementation.
- Properties may be specified in any of three ways
  - JDBC 1 connection properties passed as a java.util.Properties object (consisting of key/value pairs) as argument to DB2Driver.connect() or java.sql.DriverManager.getConnection()
  - As part of the database URL itself
    - jdbc:db2://server[:port]/databaseName[:propertyKey=value;...]
    - jdbc:db2://localhost:50000/TESTDB:user=foo;password=bar;
  - Using setXXX methods
    - most properties are defined in the abstract com.ibm.db2.jcc.DB2BaseDataSource class
- Examples of properties - ProgressiveStreaming, **securityMechanism**, loginTimeout, keepdynamic, deferPrepares, **enableSysplexWLB**, currentExplainMode, cursorSensitivity, maxTransportObjects, traceFile, traceLevel, **currentSchema**, currentSQLID, dumpPool

Connection Optimization

- All DB resources hang off Connection objects, must be managed carefully
- Creating and terminating a connection is resource consuming, both in the driver and Db2
- Client obtaining a physical database connection requires multiple network requests to
  - Handshake on DRDA protocol
  - Validate client user credentials
  - Establish code page, packet sizes etc
- Get a connection object when needed
- Reuse a connection object for multiple Statement objects when possible, use connection pooling
- Close connections promptly, don't leave connection cleanup up to garbage collection

Sysplex Workload Balancing

- Data Server Driver Type 4 supports sysplex distribution and transparent failover at transaction boundary



Use ClientInfo fields

- Can be used in WLM, RLF and profile definition and performance monitoring

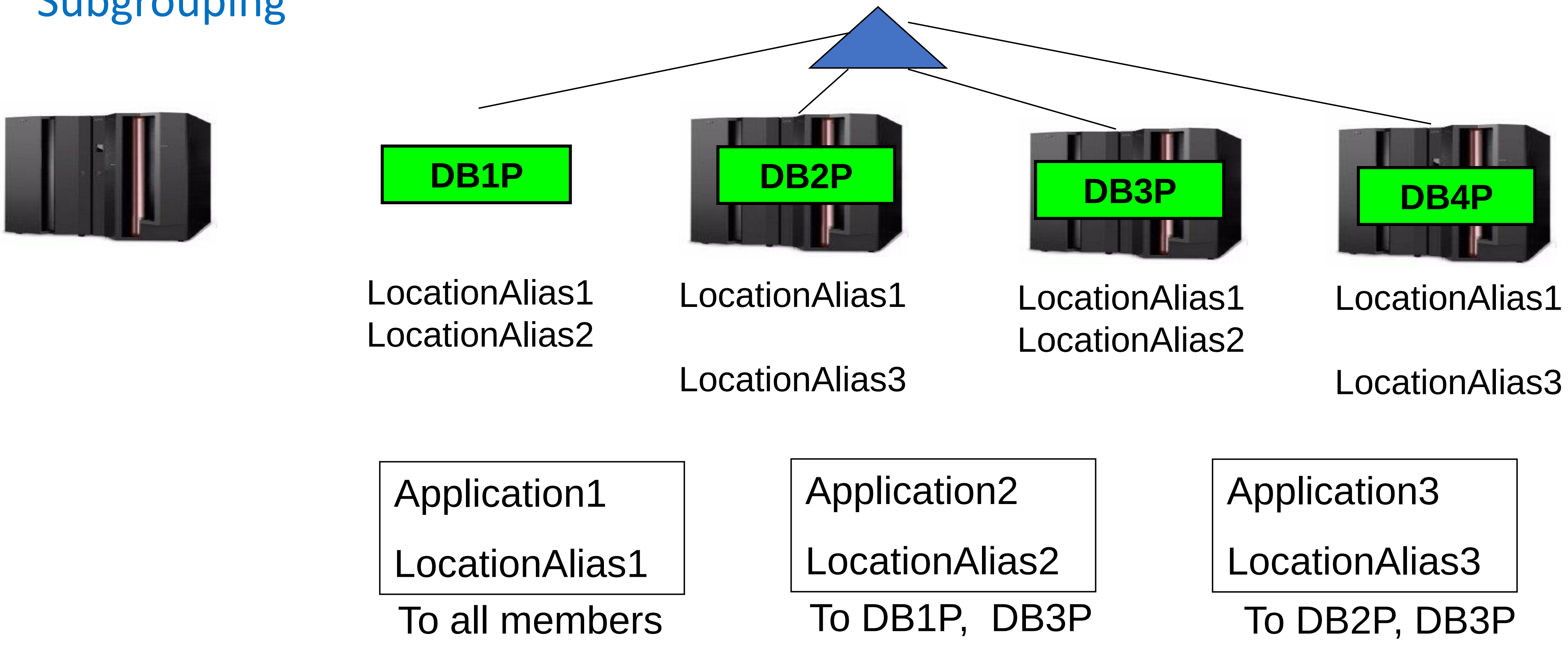
- WebSphere Application Server supports explicit and implicit setting of client information
  - Example how to call explicitly

```
...
  WSConnection conn = (WSConnection) ds.getConnection();
  props.setProperty(WSConnection.CLIENT_ID, "user123");
  conn.setClientInformation(props);
```

  - Example how to call implicitly by turning on WebSphere Trace Group

```
      WAS.clientinfo=all=enabled or
      WAS.clientinfopluslogging=all=enabled
```

Db2 Location Alias for Subgrouping



*–**MODIFY DDF** command with the **ALIAS** keyword to configure and manage aliases dynamically without taking a Db2 or DDF outage.*

Db2 High Performance DBAT

- **High Performance DBATs reduce CPU consumption by**
  - Supporting RELEASE(DEALLOCATE) to avoid repeated package allocation/deallocation
  - Avoiding connection going inactive and then back to active
  - Avoid DBAT being pooled at commit and probably reused by a different connection

- **Enable High Performance DBAT**
  - BIND client packages into different collection coll2 with RELEASE(DEALLOCATE)
  - BIND other frequently executed packages with RELEASE(DEALLOCATE)
    - E.g.trigger packages
  - Set -MODIFY DDF PKGREL(BNDOPT) to enable High Performance DBAT
  - Monitor MAXDBAT as more DBAT could stay active at any point in time and review size of application server connection pools for reasonable size

- **In WAS datasource property definition point to new collection**
  - E.g. jdbcCollection=coll2

- **JDBC Type 2 connection use RRSAF and would similarly benefit from client packages bound with RELEASE(DEALLOCATE)**

Db2 Connection Profile

- **Problem:** For distributed workloads, low priority or poorly behaving client applications may monopolize Db2 resources and prevent high-priority applications from executing.

- **Solution:** Increased granularity of monitoring for system level activities
  - Number of connections
  - Number of threads
  - Idle thread timeout

- Profiles specified in **SYSIBM.DSN_PROFILE_TABLE**
- Db2 supports filtering and threshold monitoring of system related activities via keywords
  - Number of threads - Db2 11 special register
  - Number of connections - Db2 12 Db2-provided global var
  - Idle thread timeout
- Scope filters
  - ROLE (available through Trusted Context support)
  - Product-specific identifier

Prepared Statement Objects - Benefits

- PreparedStatement objects vs. Statement objects
  - 2 DB calls needed for fetch (describe, data)
  - PreparedStmt makes description calls at construction time, Statement makes them on every execution.
  - PreparedStatement enables Statement Pooling
  - Use Statement when SQL is not executed often

- PreparedStatement object pool
  - Client side optimization
  - Pool of PreparedStatement and CallableStatement objects, not active in a Connection
  - Reduced overhead of Java object creation and garbage collection
  - Pool exists for the life of an open connection, effectiveness depends on connection pooling.
  - No impact to application

- **No concern for SQLJ**

Statement or PreparedStatement

- Statement example

```
Statement stmt = con.createStatement();
stmt.executeUpdate("INSERT INTO EMPLOYEE VALUES( 'John', 123 )");
stmt.executeUpdate("INSERT INTO EMPLOYEE VALUES( 'Mary', 425 )");
```

- PreparedStatement example

```
PreparedStatement ps = con.prepareStatement("INSERT INTO EMPLOYEE VALUES(?,
?)");
ps.setString(1,"John");
ps.setInt(2, 123);
ps.executeUpdate();
ps.setString(1,"Mary");
ps.setInt(2, 425);
ps.executeUpdate();
```

Db2 Dynamic Statement Cache

- Dynamic statement prepared at run time
- Dynamic statement cache
    - To improve performance of dynamic SQL
    - Enabled by DSNZPARM CACHEDYN = YES
    - Allows applications (multiple threads) to reuse and share prepared statements
- Prepared stmt is saved in an in-memory cache
- Subsequent prepares of same stmt loads from cache if cache match criteria met (sql, authid, special regs, bind options etc.)
- Cache pool shared by different threads, plans and packages (“global cache”)
- Good cache hit rate produces significant performance benefits
- A Full Prepare can consume 10-100X more CPU than a Short Prepare!

Dynamic Statement Cache

- SQL can be EXPLAINed using the ‘EXPLAIN STMTCACHE’ feature.
  - Populates various explain tables with details on statements in the dynamic statement cache including access path information

- Use Dynamic SQL Stmt section of statistics to monitor the Global Cache Hit Ratio % to determine if the cache size needs to be increased.

DYNAMIC SQL STMT	QUANTITY	/SECOND	/THREAD	/COMMIT
PREPARE REQUESTS	5099.0K	16.2K	N/C	7.83
FULL PREPARES	0.00	0.00	N/C	0.00
SHORT PREPARES	5098.5K	16.2K	N/C	7.83
GLOBAL CACHE HIT RATIO (%)	100.00	N/A	N/A	N/A
IMPLICIT PREPARES	0.00	0.00	N/C	0.00
PREPARES AVOIDED	0.00	0.00	N/C	0.00
CACHE LIMIT EXCEEDED	0.00	0.00	N/C	0.00
PREP STMT PURGED	0.00	0.00	N/C	0.00
LOCAL CACHE HIT RATIO (%)	N/C	N/A	N/A	N/A
CSWL - STMTS PARSED	0.00	0.00	0.00	0.00
CSWL - LITS REPLACED	0.00	0.00	0.00	0.00
CSWL - MATCHES FOUND	0.00	0.00	0.00	0.00
CSWL - DUPLS CREATED	0.00	0.00	0.00	0.00

Literal Replacement for Global Dynamic Statement Cache

- Dynamic SQL with literals can be re-used in the cache
  - Literals replaced with &
    (similar to parameter markers but not the same)
- To enable set the property enableLiteralReplacement='YES' in the JCC Driver
- Lookup Sequence
  - Original SQL with literals is looked up in the cache
  - If not found, literals are replaced and new SQL is looked up in the cache
    - Additional match on literal usability
    - Can only match with SQL stored with same attribute, not parameter marker
  - If not found, new SQL is prepared and stored in the cache
- **Db2 12 support as BIND option CONCENTRATESTMT on package**

Why SQLJ?

- Static SQL performance for Java applications
- Static SQL authorization model
- Monitoring/Manageability
  - Static SQL packages for accounting/monitoring
  - Static SQL locks in access path, so that access path changes do not occur without a conscious choice
- Measurements with the IRWW workload comparing JDBC vs SQLJ with the T2 driver

	Throughput (ETR)	Normalized Throughput (ITR)	z/OS CPU Utilization	CL.1 CPU time
JDBC T2	2636.83	3773.37	69.88	0.000672
SQLJ T2	2694.80 (+2.20%)	5174.35 (+37.13%)	52.08 (-25.47%)	0.000457 (-32.00%)

Batching INSERT and SELECT

- INSERT – JDBC API

```
ps = conn.PrepareStatement("INSERT INTO EMPLOYEE VALUES (?)")
ps.setInt(1)
ps.addBatch()
ps.setInt(2)
ps.addBatch()
Int[] returncodes = ps.executeBatch()
```

- SELECT - Db2 driver extension (executeDB2QueryBatch)

```
PreparedStatement ps = conn.prepareStatement("SELECT * from T1 where C1 = ?")
  ps.setInt(1,1)
  ps.addBatch()
  ps.setInt(1,2)
  ps.addBatch()
  ((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).executeDB2QueryBatch();
  While (ps.getMoreResults()) {
    Rs = ps.getResultSet()
    While (rs.next()){
   }
  rs.close
```

Load Data from Distributed Client

- **Current solutions**
  - First, file transfer from client to z/OS and then use Db2 LOAD utility
    - Invoke DSNUTILU stored procedure to call LOAD utility
  - Other DRDA connection-based approaches
    - Client application can issue SQL INSERTs
    - Db2 Connect IMPORT utility with SQL INSERT option and IXF input
    - Db2 z/OS cross-loader from Db2 LUW tables
- **New: DRDA fast load**
  - Easy and fast loading of data from file that resides on client
  - **Available** with **Db2 Client V11.1 FP1** and **Db2 V12R1M100**
    - **Java T4** application can use ZLOAD method in a Db2 connection class
    - Supported client file formats:
      Internal format, as well as delimited and spanned (LOB/XML data)
  - Performance results showed DRDA fast load as fast as LOAD utility
    - Significant elapsed time reduction and up to 100% zIIP exploitation

DRDA Fast Load Implementation

Data Server Client for JDBC/SQLJ

```
…
DB2Connection db2conn = (DB2Connection)con;

// LOAD statement text is in a string or input file
String loadstmt = "TEMPLATE SORTIN ….” +
"TEMPLATE SORTOUT  ….” +
"SYSCLIEN WORKDDN(SORTIN) REPLACE PREFORMAT
LOG(NO) REUSE  NOCOPYPEND FORMAT DELIMITED
UNICODE INTO TABLE MYID.CUSTOMER_DATA NUMRECS
30000";

// Name of the file that contains the input data
String dataFilename = "C:\\customer.data";

// Identifier for this run of the LOAD utility
String utilid = "ZLOADTEST1";
LoadResult lr = db2conn.zLoad(loadstmt, dataFilename, utilid);

int returnCode = lr.getReturnCode();
String loadMessage = lr.getMessage();
```

File formats
ftp block mode

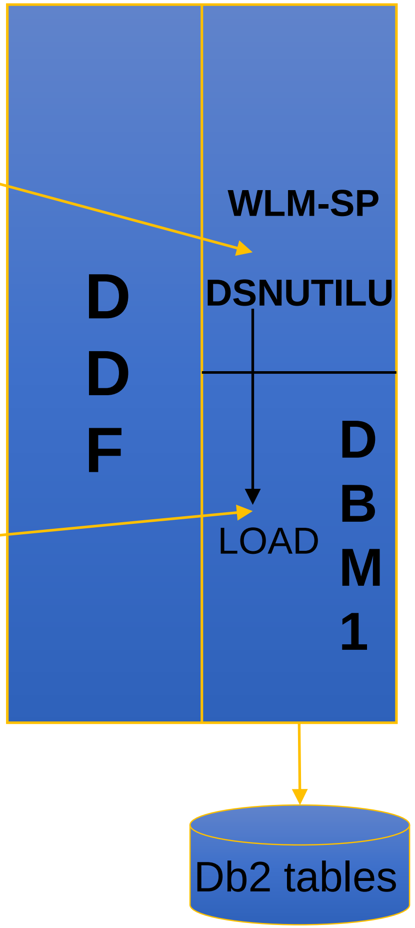
1. **INTERNAL**, used by SAP
2. **DELIMITED**
3. **SPANNED** for LOB/XML columns

Db2 z/OS server

1. Connect

2. Establish LOAD environment

3. Start data stream and provide continuous blocks for LOAD



- Cancel of LOAD with RESUME YES
→ RECP / RBDP, despite BACKOUT YES

Java Performance Problem Areas

➢ Java Application

- Autocommit(on) - default
- Mismatch of Java and Db2 data types
- Usage of String for numbers
- Retrieval of unused columns (select *)
- Transaction isolation REPEATABLE READ (default in WAS) or SERIALIZABLE
- Open cursor SELECT ... FOR UPDATE for locking semantics
  - Consider using WITH RS USE AND KEEP UPDATE LOCKS

➢ JDBC

- JDBC resources not closed (cursor, statements, connections)
- No usage of Parameter Markers
  - E.g. select c1, c2 FROM t1 WHERE c3=?
-> use literal replacement option
- Cursor are defined as hold by default
- Usage of Statement() instead of preparedStatement() objects
  - No object caching in WAS

Locking and Concurrency

- If deadlocks and timeouts, turn on Db2 Performance Trace class(6)
  - Use LOCKSIZE ROW selectively for top reported tables
  - Combine with MEMBER CLUSTER if data sharing to reduces page P-lock and page latch contention on data pages
    - Can be defined via deferred ALTER/REORG

- Review indexes
  - Missing index causing table scan and deadlocks
  - Drop unused indexes and Ris

- Zparm SKIPUNCI – skip uncommitted inserts for ISOLATION(CS|RS)

Let’s Practice

- Type 2 vs. Type 4 connectivity
  - SE utilization
  - Failover configuration considerations

```
AVERAGE           APPL(CL.1)  DB2 (CL.2)
------------      ----------  ----------
ELAPSED TIME        0.278042    0.000280
 NONNESTED          0.278042    0.000280
 STORED PROC        0.000000    0.000000
 UDF                0.000000    0.000000
 TRIGGER            0.000000    0.000000

CP CPU TIME         0.000111    0.000080
 AGENT              0.000111    0.000080
  NONNESTED         0.000111    0.000080
  STORED PRC        0.000000    0.000000
  UDF               0.000000    0.000000
  TRIGGER           0.000000    0.000000
 PAR.TASKS          0.000000    0.000000

SE CPU TIME         0.000097    0.000071
 NONNESTED          0.000097    0.000071
 STORED PROC        0.000000    0.000000
 UDF                0.000000    0.000000
 TRIGGER            0.000000    0.000000
```

Let's Practice ….

- SQL activity per commit
- Thread management per commit

NORMAL TERM.	AVERAGE	TOTAL
NEW USER	0.00	0
DEALLOCATION	0.00	7
APPL.PROGR. END	0.00	0
RESIGNON	1.00	53571
DBAT INACTIVE	0.00	0
TYPE2 INACTIVE	0.00	0
RRS COMMIT	0.00	0

SQL DML	AVERAGE	TOTAL
SELECT	0.00	0
INSERT	0.00	0
ROWS	0.00	0
UPDATE	0.00	0
ROWS	0.00	0
MERGE	0.00	0
DELETE	0.00	0
ROWS	0.00	0
DESCRIBE	0.94	50413
DESC.TBL	0.00	0
PREPARE	1.00	53571
OPEN	1.00	53571
FETCH	1.00	53571
ROWS	1.22	65594
CLOSE	0.00	0
DML-ALL	3.94	211126

Useful links

- IBM Redbook - Db2 for z/OS and WebSphere Integration for Enterprise Java Applications – http://www.redbooks.ibm.com/abstracts/sg248074.html?Open

Summary

- Business critical Java applications with Db2 for z/OS as enterprise database server have been implemented commonly and successfully for many years now

- Going through installation checklist is highly recommended prior to each implementation to ensure success
  - Communication among WAS Administrator, Db2 System Programmer, and Application Architect

- No shortcuts in respect to setup for availability
  - User sees application availability and not Db2 system availability

- Monitor and react proactively and do not wait until user complains
  - Workload behavior changes over time