ApacheCon US 2004

TU08

Perform with Apache Derby/Cloudscape

Ron Reuben

Agenda

- **Assumptions & goals**
- **Derby properties primer**
- **Derby performance issues & tuning**
- **RunTimeStatistics**
- **Tips**
- **Future work**
- **References**
- **Summary**

Assumptions & Goals

- **This is a beginner level session. You are assumed to know/have**
  - Some Java & JDBC knowledge
  - Some SQL knowledge
  - RDBMS fundamentals
  - Some exposure to or knowledge of Derby
- **Goal is to identify/learn**
  - What leads to bad performance, identify some common pitfalls and learn how to fine tune your system
  - How to use RunTimeStatistics
  - Identify some guidelines (tips) for high performance

Derby Properties Primer

- **Use properties in Derby to affect behavior of**
  - Entire system
  - A specific database
- **How to specify properties**
  - derby.properties file
  - Command line JVM option

    ```
    java –Dderby.system.home=/local1/ronr MyDerbyApp
    ```
  - In an application

    ```
    Properties p=System.getProperties();
    p.put(“derby.system.home”,”/local1/ronr”);
    ```

Derby Properties Primer…

- Using a system procedure

```
call SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY
        ('derby.locks.waitTimeout', 15);
```

- **Property precedence**
  - System-wide properties set programatically (either command line option or within JDBC application code)
  - Database-wide properties
  - System-wide properties set in derby.properties
- **With network server**
  - System-wide properties set at server
  - Database-wide properties can be set by client

Derby Properties Primer…

- **Sample properties**
  - derby.storage.pageCacheSize
  - derby.storage.pageSize
  - derby.language.logStatementText
  - derby.locks.deadlockTimeout
  - derby.locks.waitTimeout
  - derby.system.home

Performance Issues & Tuning

- **Factors that affect performance**
- **Common performance pitfalls**
- **Simple performance tuning & tweaking**

Issues - Factors Affecting Performance

- **Environment issues**
- **JVM settings**
- **Database design issues**
- **Application design issues**

Factors Affecting Performance…

- **Environment issues**
  - Memory
    - Is your system paging too much?
  - Disk
    - Disk access (I/O) times (SCSI, IDE)?
    - Disk caching?
    - Sufficient space allocated for swap/virtual memory?
  - Network
    - High network latency?

Factors Affecting Performance…

- **JVM settings**
  - Garbage collector (GC)
    - J2SE 1.4.2 includes 4 different GCs
    - Know when to choose each (advanced java users)
    - Default GC usually good for most Derby based applications
  - Heap size
    - JVM grows & shrinks the heap to stay within range specified by *–XX:min(max)HeapFreeRatio* (J2SE 1.4.2)
    - Default max heap is 64MB (J2SE 1.4.2)
    - More heap = more space for GC to scan = more “pauses” in JVM

Factors Affecting Performance…

- **JVM settings**
  - Hotspot VM
    - Which compiler system to choose?
    - “-client” system: fast startup, small footprint apps (GUIs)
    - “-server” system: when performance is of prime importance

```
$ java -server -version
      java version "1.4.2_03“
      Java(TM) 2 Runtime Environment, Standard Edition    (build
   1.4.2_03-b02)
      Java HotSpot(TM) Server VM (build 1.4.2_03-b02,    mixed
   mode)
```

Factors Affecting Performance…

- **Database design issues**
  - Non-normalized tables
    - Redundancies in table?
  - Incorrect/insufficient indexing
    - Indexes help DBMS create an optimized query plan
  - Trigger misuse
    - Larger/more complex trigger = slower Insert/Update/Delete
  - Insufficient disk storage tuning
    - Page size too small/large

Factors Affecting Performance…

- **Application design issues**
  - Bad query design (inefficient SQLs)
    - Are you using “expensive” queries?
  - Inefficient JDBC code
    - Classic case – use of *Statement* instead of *PreparedStatement*
  - Transaction isolation level
    - Affects concurrency and response time
    - Determines lock granularity chosen

Common Performance Pitfalls

- **Statement v/s PreparedStatement (JDBC)**
- **Insufficient and/or incorrect indexing**
- **Use of “expensive” queries**
- **Network Server related performance pitfalls**

Common Performance Pitfalls…

Statement v/s PreparedStatement

Pitfall – Statements

- **Use *PreparedStatement* instead of *Statement* for SQL statements that are executed many times**
- ***PreparedStatement* inherits from *Statement***
- **Instances of *PreparedStatement* contain an SQL statement that has already been compiled**
- **Precompiled SQL = significantly faster execution**
- **8-13 times (will vary) speedup with the following example (1,000 - 10,000 employee rows)**

Pitfall – Statements…

```
//connection “conn” already established
PreparedStatement ps = conn.prepareStatement(“insert into tab(empName,
   empSalary) values(?,?)”);
for (int i=0; i<numEmployees; i++) {
  //get the employee details from some other system
  try { //update the current database & table
     ps.setString(1, emplName);
    ps.setFloat(2, emplSalary)
    ps.executeUpdate();
  } catch (SQLException sqle) {…
```

Pitfall – Statements…

Instead of

```
//connection “conn” already established
Statement s = conn.createStatement();
for (int i=0; i<numEmployees; i++) {
  //get the employee details from some other system
  try { //update the current database & table
     //following statement compiles afresh each time it
     //is executed
     s.execute(“insert into tab (empName, empSalary)
            values(‘” + empName + “’,” +
            empSalary + “)”);
  } catch (SQLException sqle) {…
}
```

Common Performance Pitfalls…

Insufficient/Incorrect Indexing

Pitfall – Indexing

- **Indexing is often an easy fix to most database performance issues**
- **Indexing is only useful when a WHERE clause is used**
- **Without a WHERE clause, Derby returns all data in table**
- **Indexes are built by Derby when**
  - Primary, Unique or Foreign Key constraints are defined
  - User explicitly creates an index

Pitfall – Indexing…

- **Index useful in situations where**
  - One of the columns in WHERE clause is the 1st column in the index's key
  - All data requested by query is contained in the index
  - Derby can use the index to avoid an extra sort (ORDER BY)
- **Queries that require sorting/processing rows in descending order will benefit with an index created DESCENDING**

Pitfall – Indexing…

- **Join order can make the difference between table & index scan. For e.g.**

```
select emp.id, emp.name, emp.salary, dept.name
  from employee emp, department dept
 where emp.id = 10 and
            emp.deptId = dept.deptId
```

  - **If employee is the outer table, Derby needs to fetch potentially only 2 rows from the tables involved (assuming index on emp.id & dept.deptId exists)**
  - **If department is chosen the outer table, Derby needs to first join both tables, then filter the emp.id of "10"**
  - **If an index is provided, optimizer will usually take care of this for you**

Pitfall – Indexing…

- Know optimizer choices to help you choose better indexes
  - Index selection for search arguments (WHERE clause)
  - Join order & index selection for joins
  - Covered queries
  - Sort avoidance
  - Etc…

Common Performance Pitfalls…

Use Of Expensive Queries

Pitfall – Expensive Queries

- **Performance "nightmare" queries**

  select * from HugeTable

  ORDER BY nonIndexedColumn

  and

  select DISTINCT nonIndexedColumn

  FROM HugeTable

- **Avoid such "table scan" queries and huge sorts**
- **If Derby issues a table lock, other apps needing access to the table can block**
- **Try to use a WHERE clause on an indexed column**

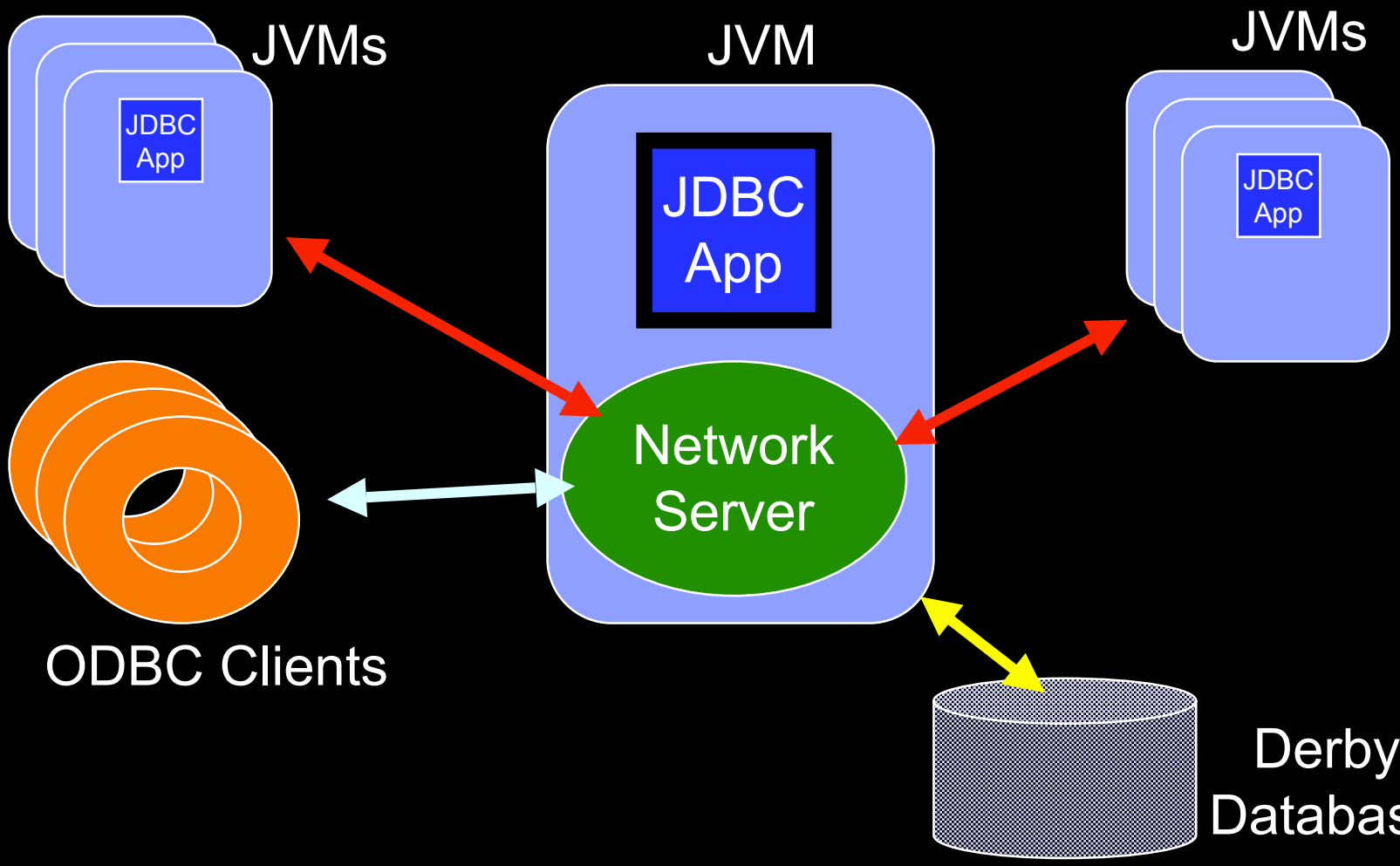
Common Performance Pitfalls…

Network Server Related

Pitfalls – Network Server Specific

- **Close your PreparedStatements**
- **By design, a close() releases server stored information on PreparedStatements in use**
- **If not, you are inducing a 'memory leak' on the server! Server can slow down due to memory bloat and/or run out of memory!!**
- **Start network server in the JVM with the heaviest database access**
- **Set derby.drda.startNetworkServer=true**

Pitfalls – Network Server Specific…



Simple Performance Tuning

- **Database page size**
- **Caches**
- **Database booting & class loading**
- **Transaction isolation level & locking**

Simple Performance Tuning…

- **Not that much "tuning" you can do as compared to enterprise level databases ("zero admin" focus!)**
- **Can tune pageSize (Derby will automatically choose 32K if larger than default 4K is required, but you can override)**
- **Can tune cache sizes**
- **Tune your transaction isolation & locking strategy**

Simple Performance Tuning…

Database Page Size

Tuning – Database Page Size

- **Tuned with the property *derby.storage.pageSize***
- **Default 4K, can be 8K, 16K or 32K**
- **Go with higher values of pageSize if**
  - You have large tables (> 10K rows)
  - You store large objects
- **Idea is to reduce I/O for large rows, tables and read-only applications**

Tuning – Database Page Size…

- **Guideline - At least 10 average sized rows per page**
- **Scenario – 1 Large column, several small**
  - Put large column at end (will overflow if required)
- **Scenario – Selective Queries, using an index**
  - Do not use large page size, potentially longer I/O
- **Scenario – Limited memory, disk space**
  - Larger pageSize may not be the best choice

Simple Performance Tuning…

Caches

Tuning – Caches

- **3 caches in Derby**
  - Page (data) cache
    - Holds data that user would access from tables & indices
  - Data dictionary cache
    - Holds information stored in the system tables
  - Statement cache
    - Holds compiled database specific statements (including PreparedStatements)
- **"Prime" caches in background for better performance**

Tuning – Data Page Cache

- **Can tune data page cache with the *derby.storage.pageCacheSize* property**
- **Defines size in number of pages**
- **Default 1000 pages, min is 40**
- **Actual memory consumed depends on page size**
- **Remember to allocate more heap if you increase page cache**

Simple Performance Tuning…

Database Booting &

Class Loading

Tuning – Class Loading & DB Booting

- **In Derby, Class loading happens**
  - When Derby boots (load the embedded driver)
  - When the first database boots
  - When the first query is compiled
- **In multi-user systems, where startup cost is acceptable**
  - Boot one or all databases at startup
  - Prepare statements at startup in a separate thread

Simple Performance Tuning…

Transaction Isolation Levels

& Locking

Tuning – Transaction Isolation Levels

- **4 isolation levels defined in Derby**
  - TRANSACTION_READ_UNCOMMITTED
  - TRANSACTION_READ_COMMITTED
  - TRANSACTION_REPEATABLE_READ
  - TRANSACTION_SERIALIZABLE
- **Tradeoff between concurrency & consistency**
- **Affects the amount of rows locked & lock decisions made by optimizer**

Tuning – Locking

- **Locking problems can hurt performance**
- **derby.locks.waitTimeout/deadlockTimeout properties provided for you to tune**
- **Force lock escalation with LOCKSIZE clause (ALTER TABLE)**
- **Derby's first goal is concurrency, so default choice is row-level locking (overheads involved)**
- **Lock escalation done if too many row locks or based on isolation level**
- **derby.locks.escalationThreshold property determines this threshold**

RunTimeStatistics

RunTimeStatistics

- **Language level tool**
- **Helps user evaluate performance & execution plan of statements**
- **Need to "set" the RunTimeStatistics attribute for a connection**

```
ij> connect 'jdbc:derby:testdb';
ij> CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
```

- **Once turned on, Derby maintains execution plan info for each statement executed within the connection**

RunTimeStatistics…

- **COMMIT ignored by RunTimeStatistics**
- **For most recently executed query, RunTimeStatistics returns**
  - Length of compile and execute time
  - Statement execution plan
- **Note – you need to turn on statistics timing to get timing information**

```
ij> CALL SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1);
```

RunTimeStatistics…

- **To access RunTimeStatistics information**

  ij> select * from employee;

  ij> VALUES
  SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS();

- **Note – If you are using RunTimeStatistics in ij, set "MaximumDisplayWidth" to prevent truncation of output**

  ij> MaximumDisplayWidth 5000;

RunTimeStatistics…

- **Statement execution plan describes**
  - Was an index used
  - If yes, then the start/stop position for index scan
  - What join order was used
  - How many rows qualify at each “node”
  - How much time was spent at each “node”
- **Can help you determine**
  - If query needs to be rewritten
  - If additional indexes are required

RunTimeStatistics…

- **Statement execution plan = tree of result set ‘nodes’**
- **A ‘node’ represents the evaluation of one portion of the statement**
- **Following query involves only one node called a *Table Scan ResultSet***

```
select * from employee
```

- **This query involves 2 nodes, a *Table Scan ResultSet* and a *Project-Restrict ResultSet***

```
select emp_id from employee
        where emp_name = ‘Ron Reuben’
```

RunTimeStatistics…

- **Self-guided example/demo**
  - See Speaker Notes attached with this page

Tips

- **Think JDBC**
  - Connection pooling
  - Batch updates
  - Use the right getXXX methods
- **Performance debugging**

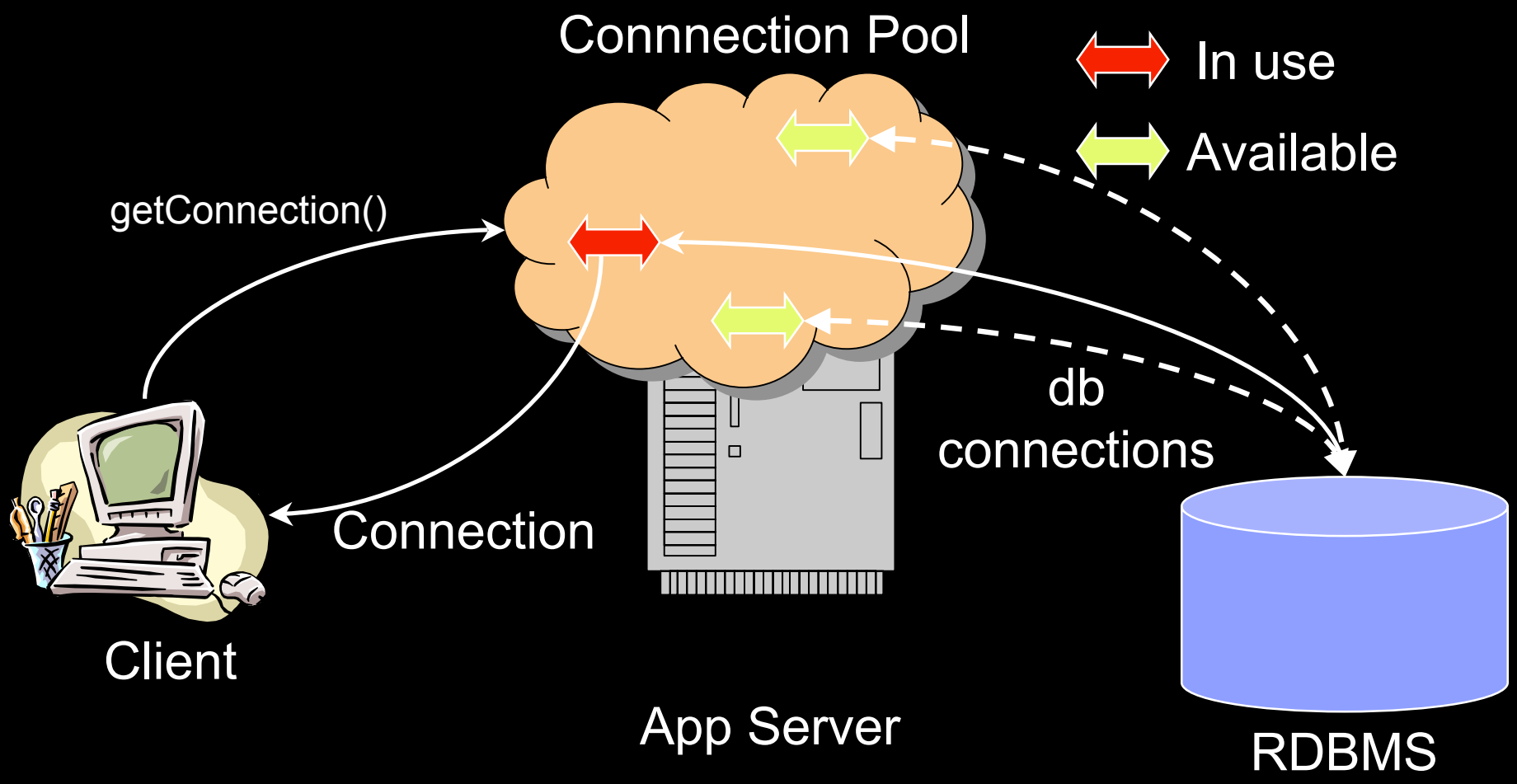
Tips…

Connection Pooling

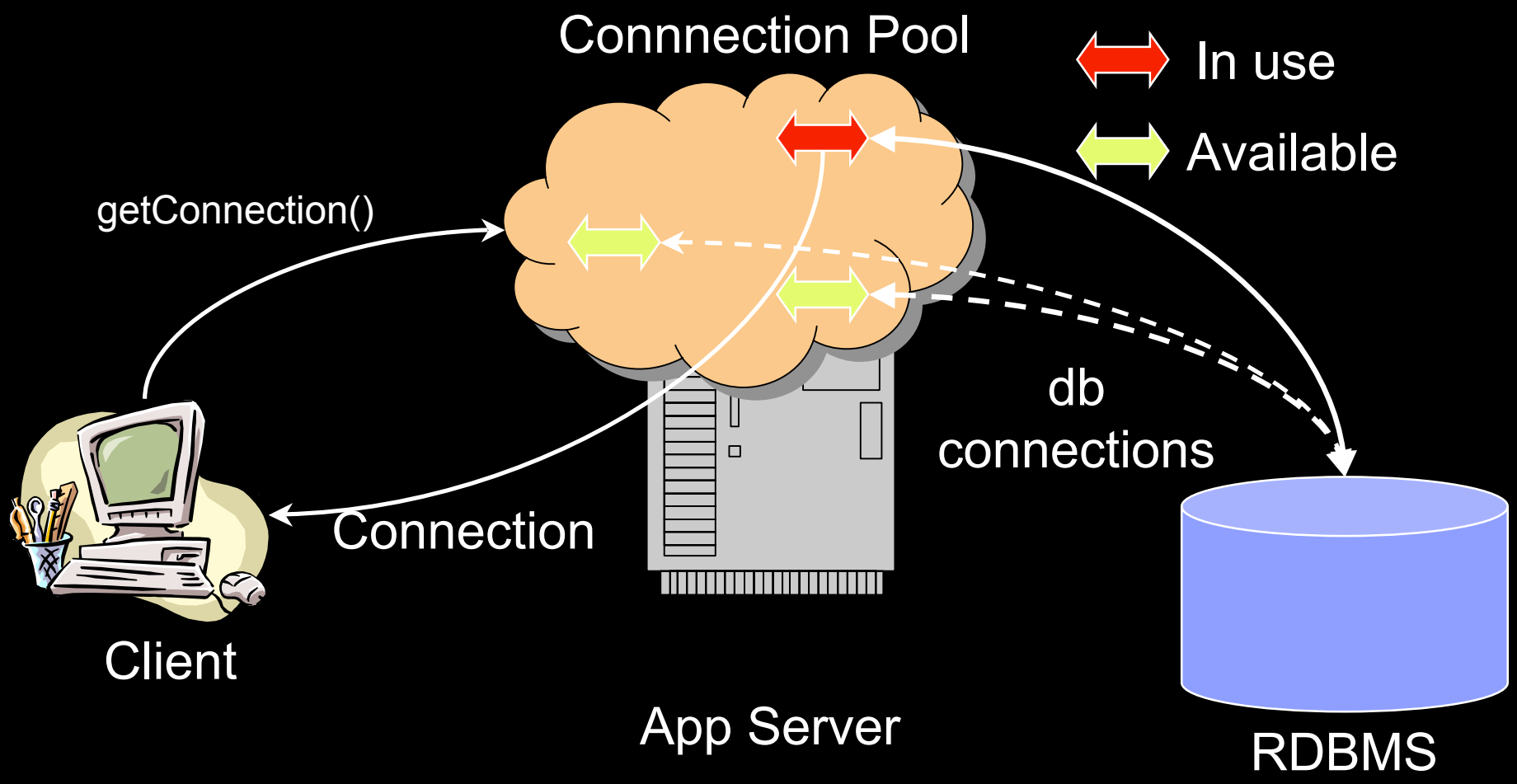
Tips – Connection Pooling

- **Establishing a connection to a database is a costly process, can degrade performance in a busy system**
- **Connection Pooling is defined and configured at the middle tier. Manager is typically an App Server**
- **Don't try to manage connections if Connection Pooling is available**
- **Application needs to use *DataSource* rather than *DriverManager* to get a connection**

Tips – Connection Pooling…



Tips – Connection Pooling…



Tips…

Batch Updates

Tips – Batch Updates

- **Send set of update statements to database for processing as a batch**
- **Can be more efficient (minimize network latency)**
- **Only statements that produce an update count can be batched. No *ResultSet* generating statements**
- **Add statements to a batch with *addBatch* method**
- **DBMS executes statements in the order added to batch; returns an array of integer update counts**
- ***BatchUpdateException* thrown if there is an error**

Tips – Batch Updates…

```
conn.setAutoCommit(false);

PreparedStatement ps = conn.prepareStatement("insert into tab(id,
   lname) values(?,?)");

for (int i = 0; i < numEmp, i++ ) {
   //get values of empid and empIname
   ……
   ps.setInt(1, empid);
   ps.setString(2, empIname);
   ps.addBatch();

}

int[] updates = ps.executeBatch();

conn.commit();
```

Tips…

Use Appropriate getXXX Methods

Tips – Appropriate getXXX Methods

- **JDBC allows multiple get methods for most datatypes**
- **E.g.**
  to "get" an integer from a *ResultSet*, you can use *getLong(), getFloat(), getDouble()* etc.
  Recommended method is *getInt()*
- **Similarly, you can use *getObject()* to retrieve all datatypes (except Boolean)**
- **Use the "recommended" get method to minimize performance impact**

Tips…

Performance Debugging

Tips – Performance Debugging

- **Environment, JVM issue?**
- **Poor application/database design?**
- **If this fundamental analysis fails, you may require debug tools**
  - jvmstat, jvmps, visualgc: Helps analyze JVM issues. Provide access to light weight performance/config instrumentation exported by HotSpot JVM
  - RunTimeStatistics: Helps analyze poor performing queries
  - Professional performance debug tools?

Tips – Performance Debugging…

- **Other things to do**
  - Effective logging with *System.out* and *System.err*
  - Log Resources: opened, closed, accessed (where appropriate)
  - Log performance timings for suspect queries

```
…
long queryStartTime = System.currentTimeMillis();
//insert suspect query here
long queryTime = System.currentTimeMillis() -     queryStartTime;
System.out.println("Suspect query took " +
                                queryTime + " msec");
…
```

Future Work

- **Potential performance enhancements for community to work on**
  - Get RunTimeStatistics to output in XML so that it can be interfaced into other tools
  - Implement the hints *Statement.setFetchSize()* and Statement.*setFetchDirection()* in the JDBC driver
  - Implement Connection Pooling interfaces in Derby code
  - Enhanced self-tuning abilities
  - Enhanced documentation

References

- **Tuning Derby**
  http://incubator.apache.org/derby/manuals/tuning/perf02.html
- **Derby Reference Manual**
  http://incubator.apache.org/derby/manuals/reference/sqlj02.html
- **Java Platform Performance: Strategies and Tactics**
  by Steve Wilson & Jeff Kesselman
  Published by Addison-Wesley Professional
  http://java.sun.com/docs/books/performance/

References

- **“jvmstat” tools**
  http://developers.sun.com/dev/coolstuff/jvmstat/
- **JDBC technology web pages**
  JDBC specs, FAQs, Documentation etc.
  http://java.sun.com/products/jdbc/

Summary

- **To ensure high performance with Derby**
  - Index, Index Index!
  - Use *PreparedStatement* instead of *Statement*
  - Avoid expensive queries, use WHERE clauses
  - Consider Connection Pooling and Batch Updates if appropriate
  - Use the recommended getXXX methods
  - Learn the choices made by the Derby optimizer to help you make better index & lock choices

Summary…

- **General observations**
  - RunTimeStatistics is a handy tool to analyze query performance
  - Only a few parameters can be “tuned” in Derby
  - When debugging performance issues, start with a broad perspective and narrow down until you reach the bottleneck (or bug)
  - Various factors could potentially affect performance, don’t be too quick to blame Derby code!