

Java DB Performance

Olav Sandstå

Sun Microsystems, Trondheim, Norway

Submission ID: 860

JAZZ00N07

THE INTERNATIONAL CONFERENCE ON JAVA TECHNOLOGY
JUNE 24 - 28, 2007 ZURICH

Java™ DB



AGENDA

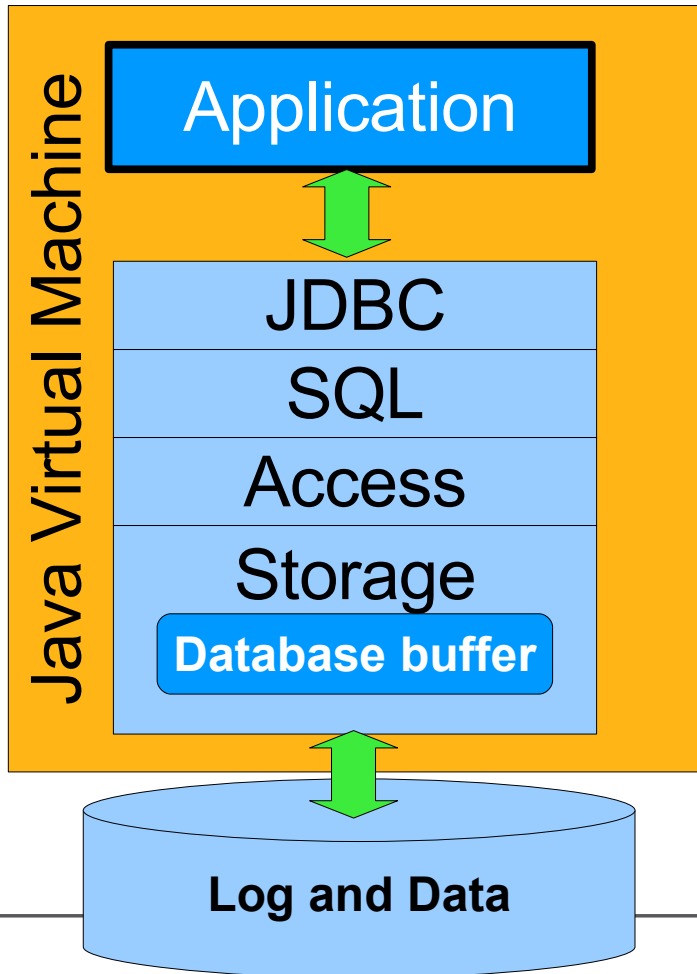
- > Java DB introduction
- > Configuring Java DB for performance
- > Programming tips
- > Understanding Java DB performance
- > Open-source database performance

Java DB

- > Sun's supported distribution of Apache Derby
 - all development done in the Apache Derby community
- > Complete SQL database including:
 - views, triggers, stored procedures, foreign keys
 - multi-user transaction support
- > Security:
 - data encryption, client authentication, GRANT/REVOKE
- > Standard based:
 - JDBC 4.0 and SQL92/99/2003/XML
- > 100% Java, bundled in Sun JDK 6 and Glassfish
- > **The database** for Java applications

Java™ DB

Java DB Architecture: Embedded



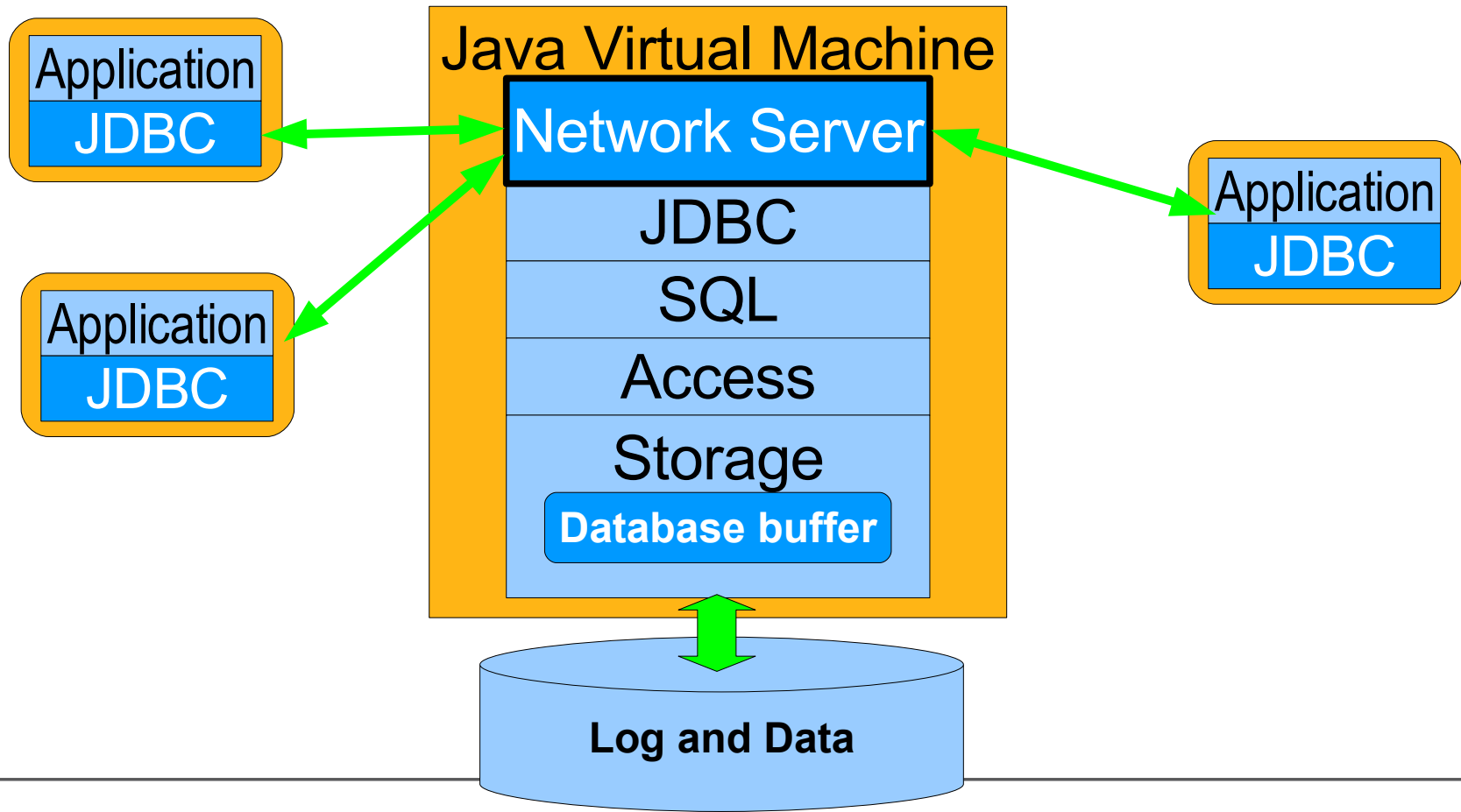
- > Include `derby.jar` in your classpath
- > Boot the Java DB engine¹⁾

```
Class.forName(  
    "org.apache.derby.jdbc.  
    EmbeddedDriver");
```
- > Create a new database

```
Connection conn =  
    DriverManager.getConnection(  
        "jdbc:derby:dbName; " +  
        "create=true");
```

1) Optional when running with JDK 6

Java DB Architecture: Client-Server



AGENDA

- > Java DB introduction
- > **Configuring Java DB for performance**
- > Programming tips
- > Understanding Java DB performance
- > Open-source database performance

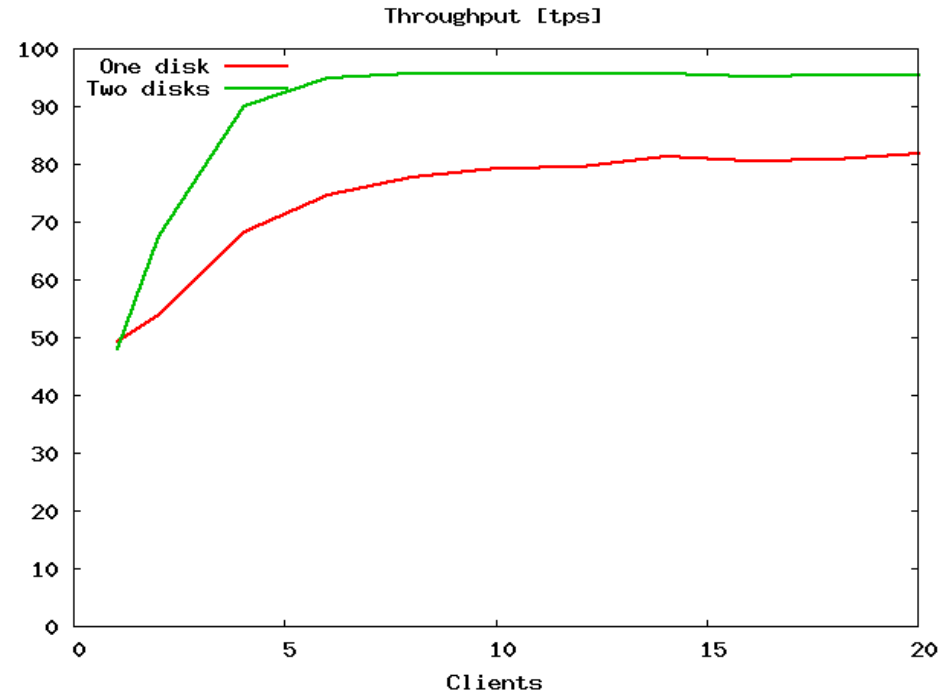
Performance Tip 1: Separate Data and Log Devices

Log on separate disk:

- > Utilize sequential write bandwidth on disk
- > Configuration:

JDBC connection url:

```
logDevice=<path>
```

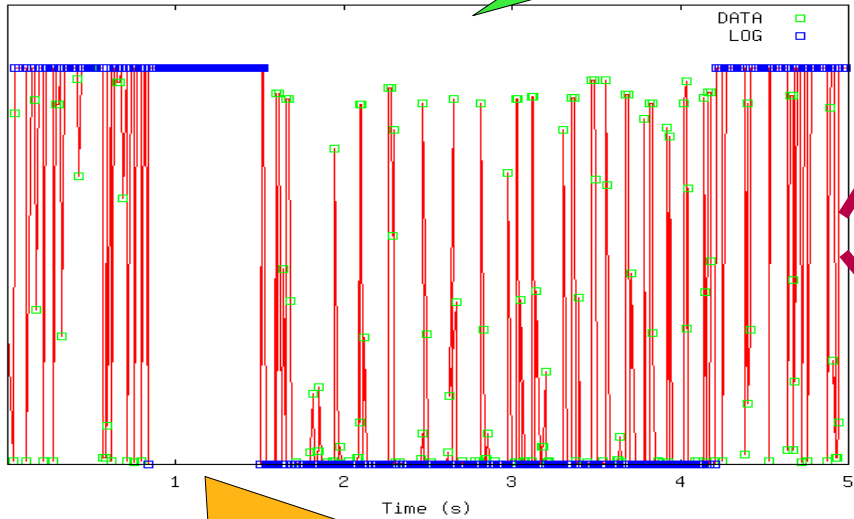


Performance tip:

- > Use separate disks for data and log device

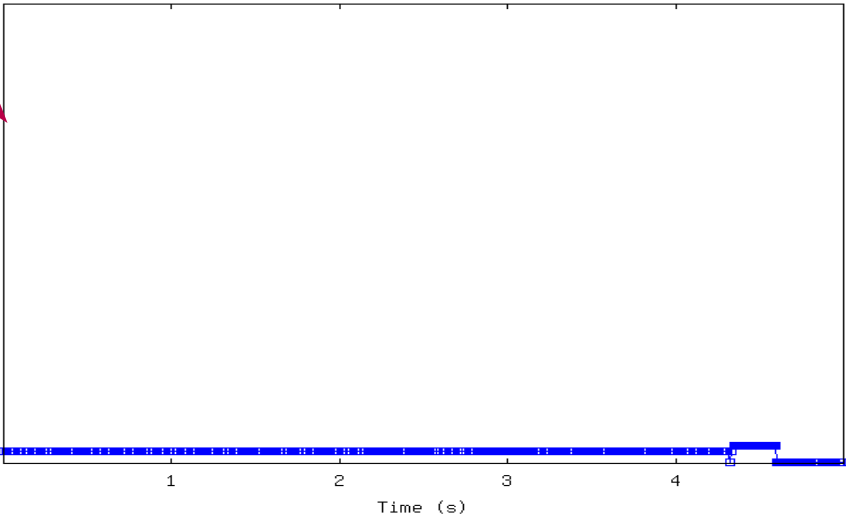
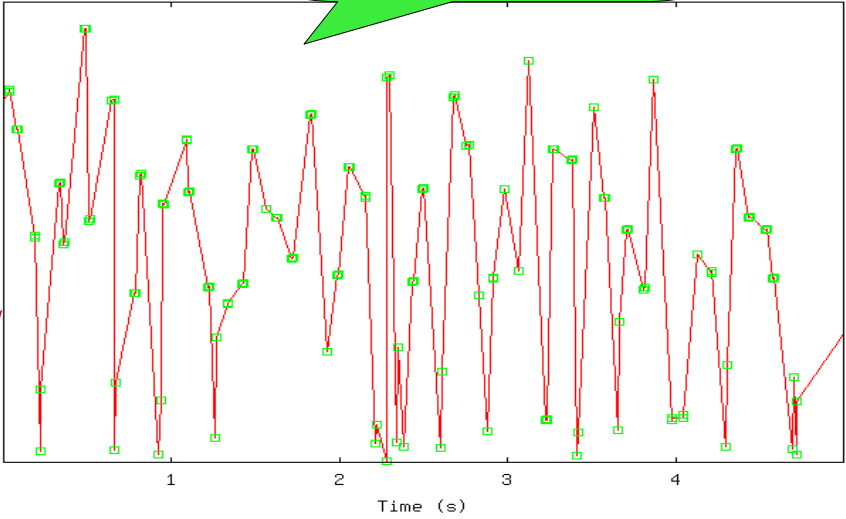
Disk Activity

Data and log on same disk



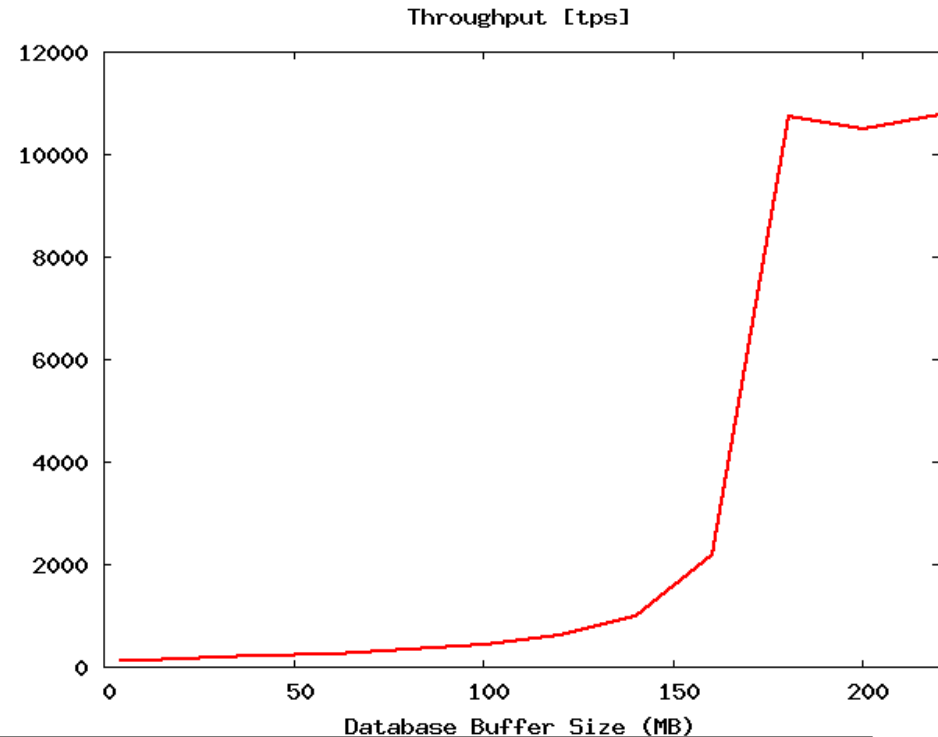
Disk head movement for 5 seconds of database activity

Data and log on separate disks



Performance Tip 2: Tune Database Buffer Size

- > Cache of frequently used data pages in memory
- > Cache-miss leads to read from disk
- > Size:
 - default 4 MB
 - `derby.storage.pageCacheSize`



Performance tip:

- > Increase the size of the database buffer to get frequently accessed data in memory

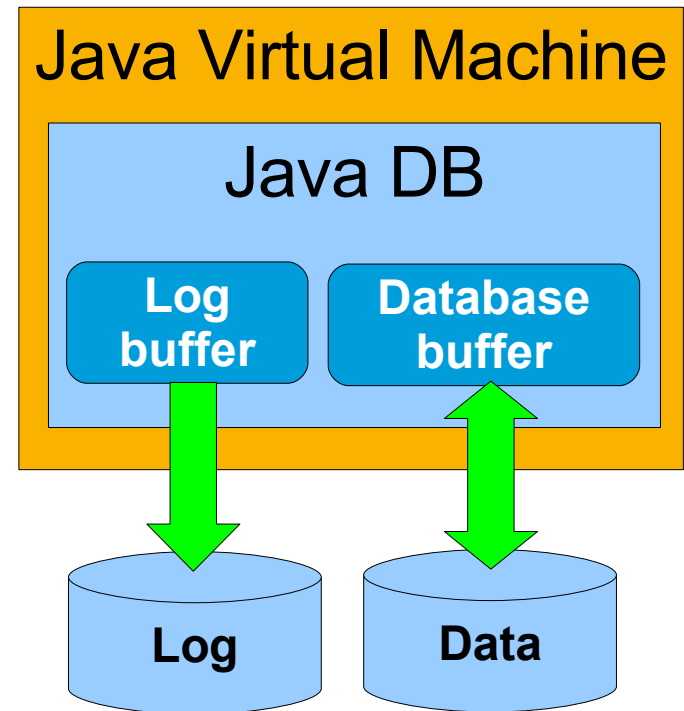
Performance and Durability: Data and Log Devices

Log device:

- > Sequential write of transaction log
- > Synchronous as part of commit
- > Group commit

Data device:

- > Data regularly written to disk as part of checkpoint
- > Data read from disk on demand



Performance Tip 3: Trade Durability for Performance

Log device configuration:

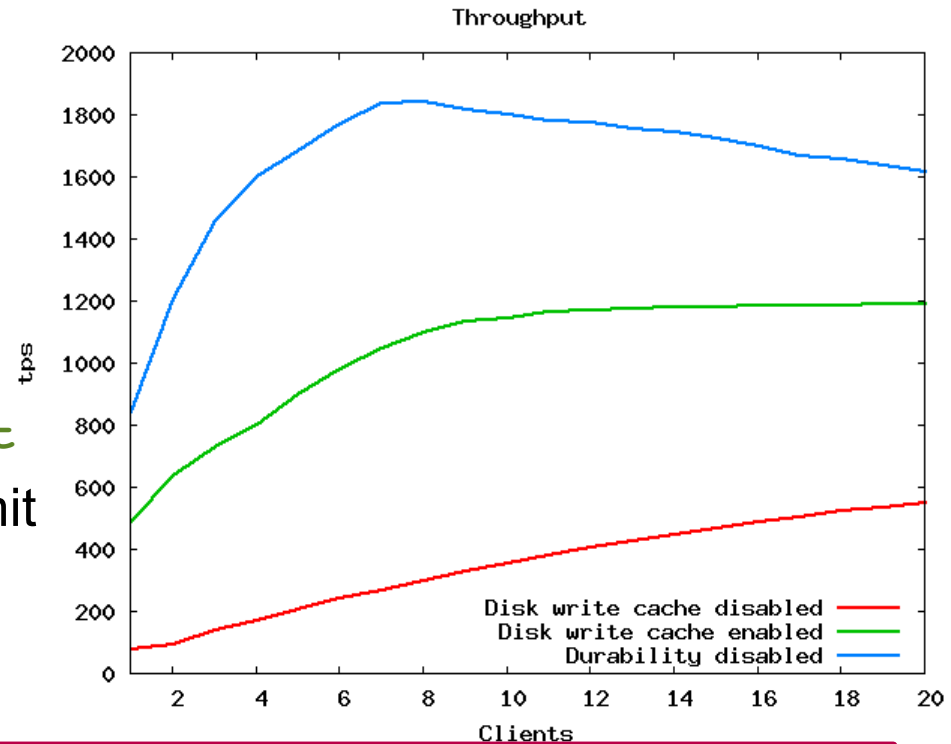
> Disk's write cache:

- disabled
- enabled

> Disable durability:

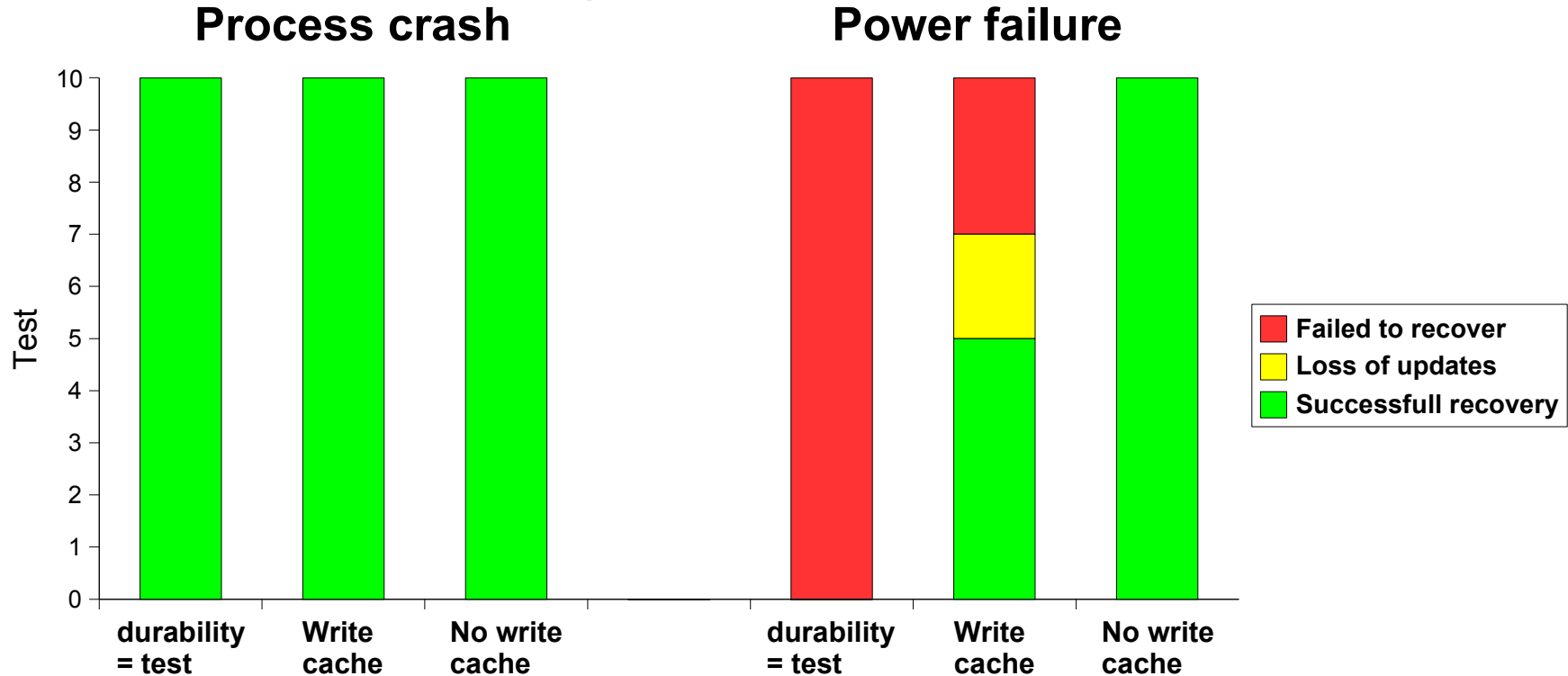
`derby.system.durability=test`

- log flushed to disk **after** commit



WARNING: Write cache reduces probability of successful recovery after power failure

Log Device Configuration: Effect on Durability



Durability tip:

Disable the disk's write cache on the log device

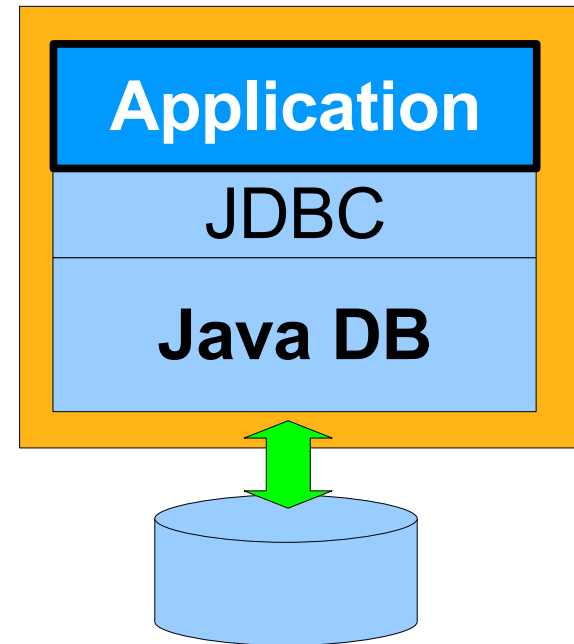
Performance Tip 4: Use Embedded Java DB

Performance advantages:

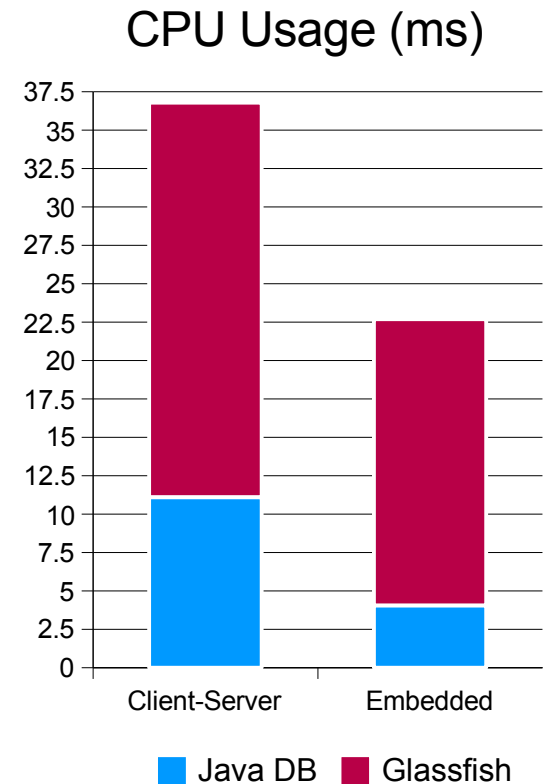
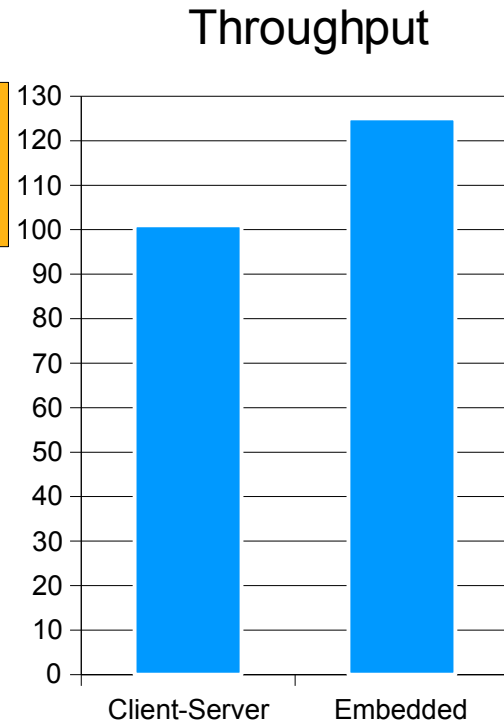
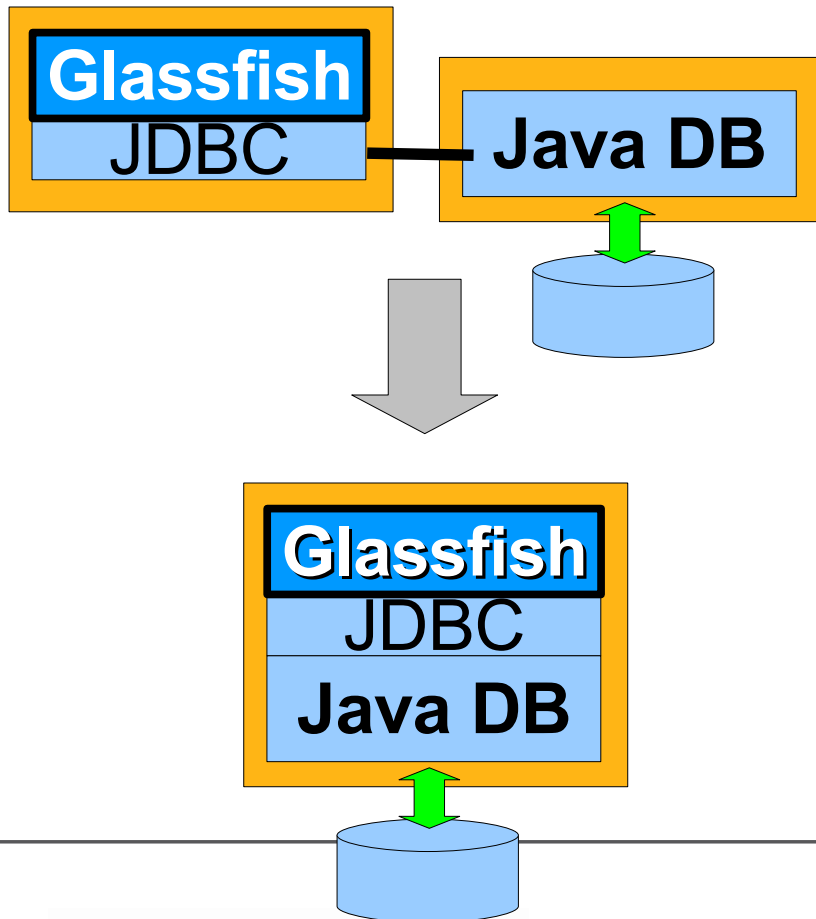
- > saves inter-process or server communication
- > reduces CPU usage
- > reduces hardware cost

Potential issues:

- > scalability (one machine)
- > JVM configuration



Glassfish and Java DB: Client-Server vs Embedded, example



AGENDA

- > Java DB introduction
- > Configuring Java DB for performance
- > **Programming tips**
- > Understanding Java DB performance
- > Open-source database performance

Performance Tip 5: Use Prepared Statements

- > Compilation of SQL statements is expensive:

```
Statement s = c.createStatement();  
while (...) {  
    s.executeQuery("SELECT * FROM t WHERE a = " + id);  
}
```

- generates Java byte code and loads generated classes

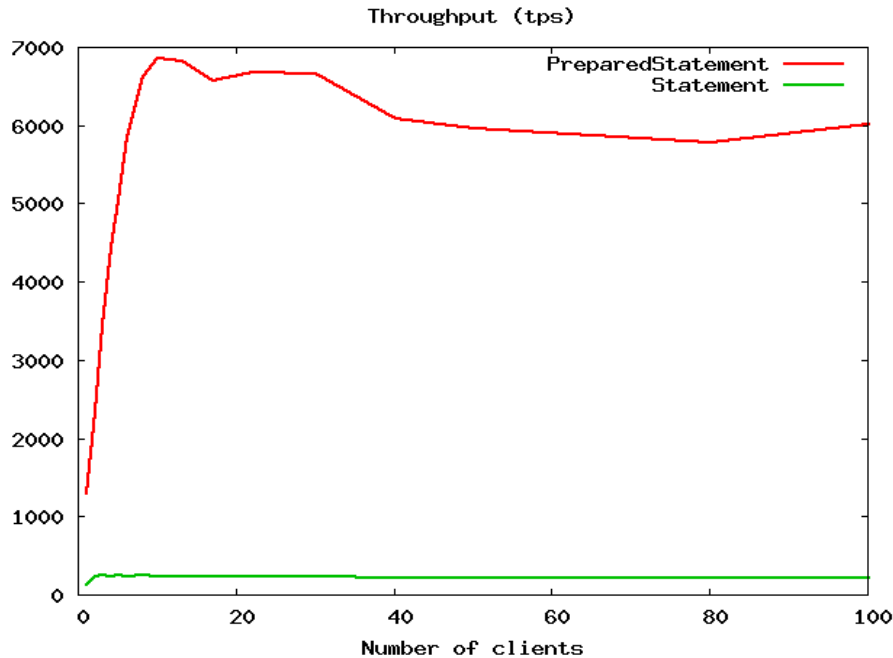
- > Prepared statements eliminate this cost:

```
PreparedStatement s =  
    c.prepareStatement("SELECT * FROM t WHERE a = ?");  
while (...) {  
    s.setInt(1, id);  
    s.executeQuery();  
}
```

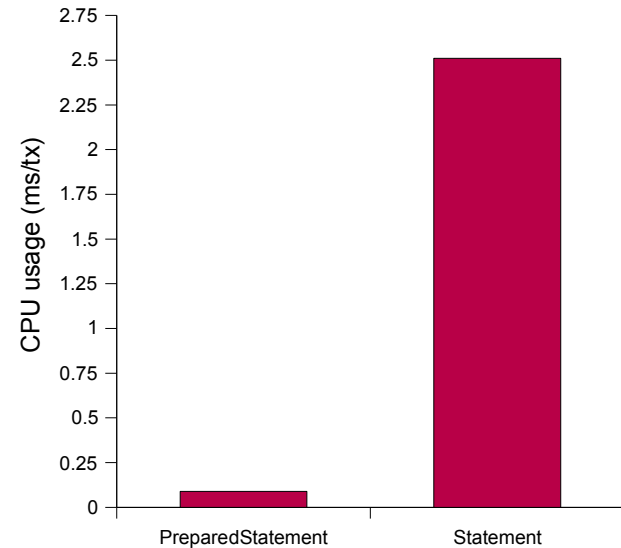
- generated Java byte code can be JIT compiled

Use Prepared Statements, example

Throughput:



CPU usage:



Performance tip:

> **USE** prepared statements - and **REUSE** them

Performance Tip 6: Avoid Table Scans

Two ways of locating data:

- > Table scan: reads the entire table
- > Index: finds the data by reading a few blocks

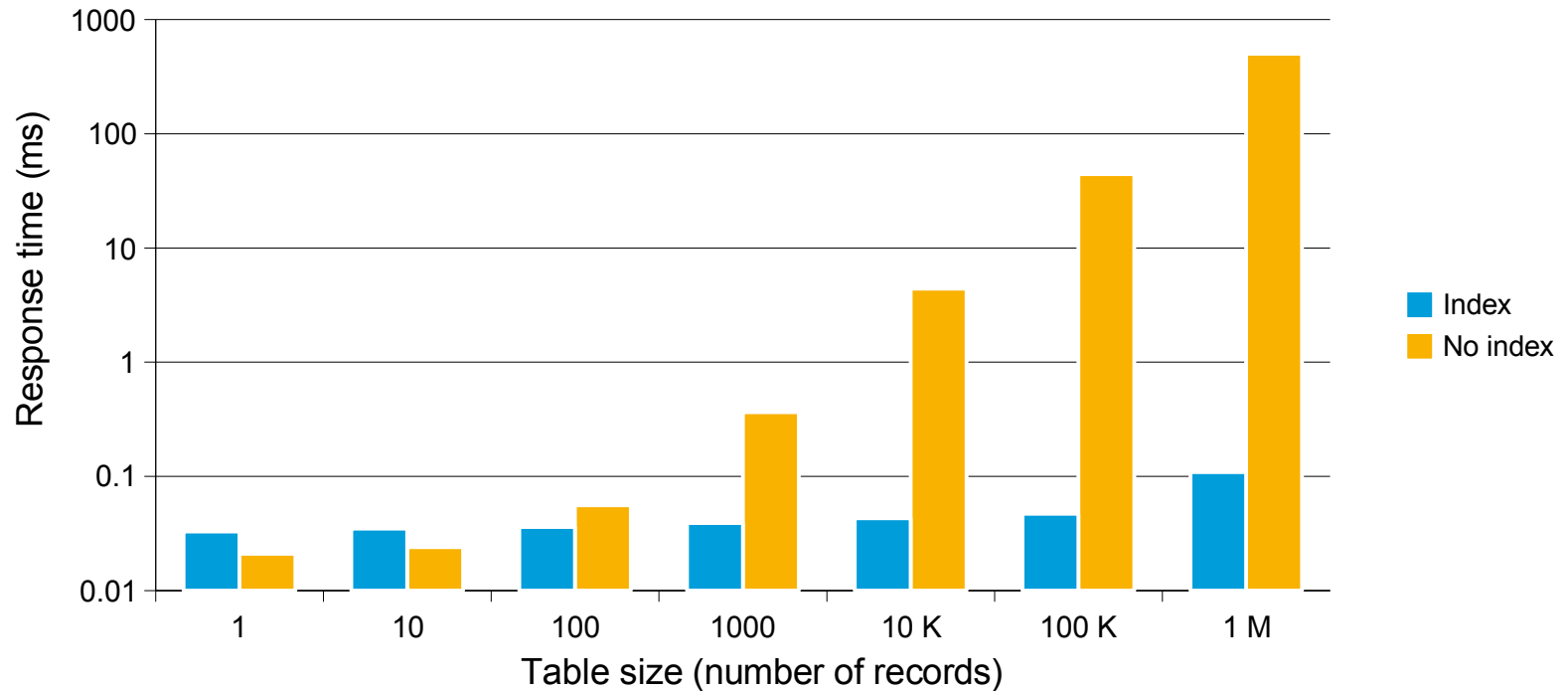
Avoid table scans:

- > Use indexes to optimize frequently used access paths:

```
CREATE INDEX indexName ON tableName (column)
```

Avoid Table Scans, example

Retrieval time for one record



Performance tip:

> Create and use indexes

Performance Tip 7: Help the Database to Perform

JDBC tips:

- > Close JDBC objects after in use
 - Connections, Statements, ResultSets, Streams
- > Use transactions - do not rely on auto-commit
 - Particularly for insert/update/delete operations
- > Batch updates reduce network traffic

AGENDA

- > Java DB introduction
- > Configuring Java DB for performance
- > Programming tips
- > **Understanding Java DB performance**
- > Open-source database performance

Performance Tip 8: Know the Load and the Resource Usage

- > Know the load on the database:
 - `derby.language.logStatementText=true`
 - all executed queries written to derby.log
- > Know how the queries are executed:
 - `derby.language.logQueryPlan=true`
- > Query-plan and run-time statistics:
 - `SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1)`
 - `SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1)`
 - `SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS()`
- > Use OS and Java tools to find resource usage:
 - CPU, memory, disk IO for log and data device

Performance tip:

- > Use the available tools to understand what the database is doing and where resources are spent

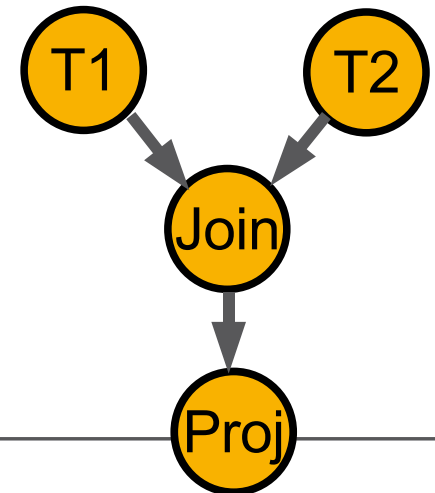
Example: Query Plan and Run-time Statistics

```
// enable run-time statistics
Statement s = c.createStatement();
s.executeUpdate("CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1)");
s.executeUpdate("CALL SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1)");

// execute query
ResultSet rs = s.executeQuery("SELECT T1.c2 from T1, T2
                               where T1.c2 = T2.c2 and T1.c2 < 800");

while (rs.next()) {}
rs.close();

// retrieve query plan and run-time statistics
rs = s.executeQuery("VALUES
                    SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS()");
rs.next();
String str = rs.getString(1);
System.out.println("Query Plan: " + str);
```



Performance Tip 9: Optimizer Overrides

> Override execution strategy selected by optimizer

> Force use of specific index:

```
SELECT * FROM t1 --DERBY-PROPERTIES index=t1_c1
WHERE c1=1
```

> Force use of constraint:

```
SELECT * FROM t1 --DERBY-PROPERTIES constraint=c
WHERE c1=1 and c2=3
```

> Force specific JOIN order and JOIN strategy:

```
SELECT * FROM --DERBY-PROPERTIES joinOrder=FIXED
t1 , t2 --DERBY-PROPERTIES joinStrategy=NESTEDLOOP
WHERE t1.c1=t2.c1
```

– Join strategies: HASH and NESTEDLOOP

Optimizer Overrides Example

```
SELECT t1.c2 FROM --DERBY-PROPERTIES joinOrder=FIXED
t1, t2 --DERBY-PROPERTIES joinStrategy=NESTEDLOOP
WHERE t1.c2 = t2.c2
```

Optimizer override	Estimated cost	CPU (ms)
None	2311	12.0
joinOrder=FIXED	2505	12.0
joinStrategy=NESTEDLOOP	3404	14.7
FIXED and NESTEDLOOP	3404	14.4

Performance tip:

> Use optimizer overrides - **but only** when needed

Performance Tip 10: Understand Locking Issues

- > Lock based concurrency control
- > Isolation level:
 - Reducing isolation level increases concurrency
- > Lock escalation:
 - Default: escalation from **row locks** to **table locks** when 5000 locks are set on the table
 - `derby.locks.escalationThreshold=100`
 - `LOCK TABLE t1 IN {SHARE|EXCLUSIVE} MODE`
- > Deadlock tracing:
 - `derby.locks.monitor=true`
 - `derby.locks.deadlockTrace=true`

Understand Locking Issues

Retrieve lock information:

```
SELECT * FROM SYCS_DIAG.LOCK_TABLE
```

XID	TYPE	MODE	TABLENAME	LOCKNAME	STATE	INDEXNAME
186	ROW	X	T2	(1, 9)	GRANT	
184	ROW	S	T2	(1, 9)	WAIT	
188	ROW	X	T1	(1, 11)	GRANT	
186	ROW	S	T1	(1, 11)	WAIT	
186	ROW	S	T1	(1, 1)	GRANT	SQL070425023
188	ROW	S	T1	(1, 1)	GRANT	SQL070425023
184	ROW	X	T1	(1, 7)	GRANT	
188	ROW	S	T1	(1, 7)	WAIT	
186	TABLE	IX	T2	Tablelock	GRANT	
184	TABLE	IS	T2	Tablelock	GRANT	

AGENDA

- > Java DB introduction
- > Configuring Java DB for performance
- > Programming tips
- > Understanding Java DB performance
- > **Open-source database performance**

Java DB 10.3: Performance Improvements

Embedded:


- > reduced synchronization & context switches
- > reduced CPU usage
- > reduced number of disk updates to log device
- > concurrent read/writes on data device

Client-server:

- > improved streaming of LOBs

SQL Optimizer:

- > improved optimization

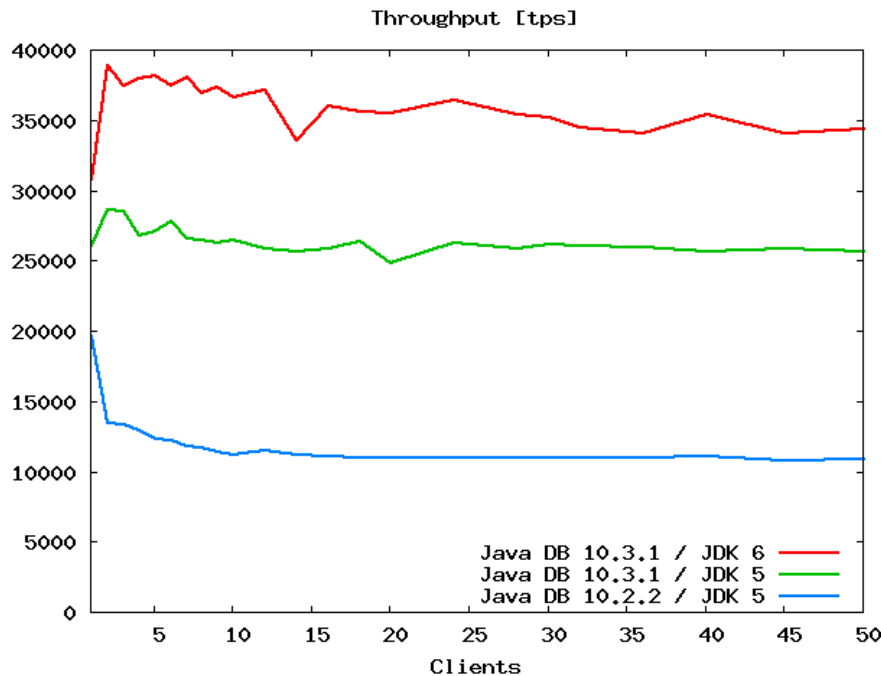


30-200% increased
throughput on
simple queries

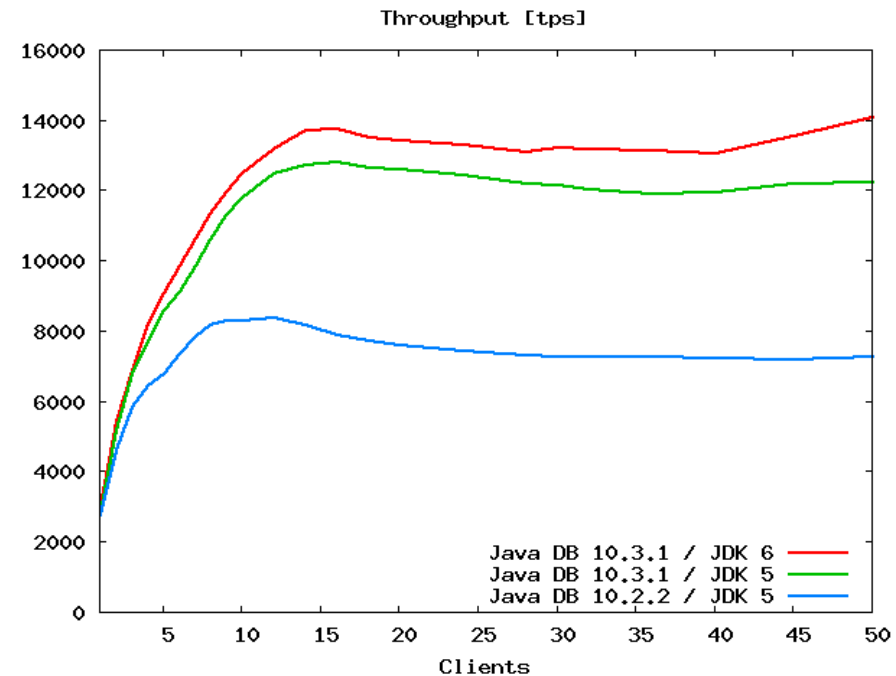
Performance Improvement, example

Upgrading to Sun JDK 6 and Java DB 10.3.1 beta:

Embedded:



Client-server:



Load: Select one record in a table

Open-Source Database Performance

Databases:

- > Java DB 10.3.1 beta
 - embedded
 - client-server
- > PostgreSQL 8.1.8
- > MySQL 5.0.33
 - with InnoDB



Load clients:

1. Select load:

1 single-record select

2. Update load:

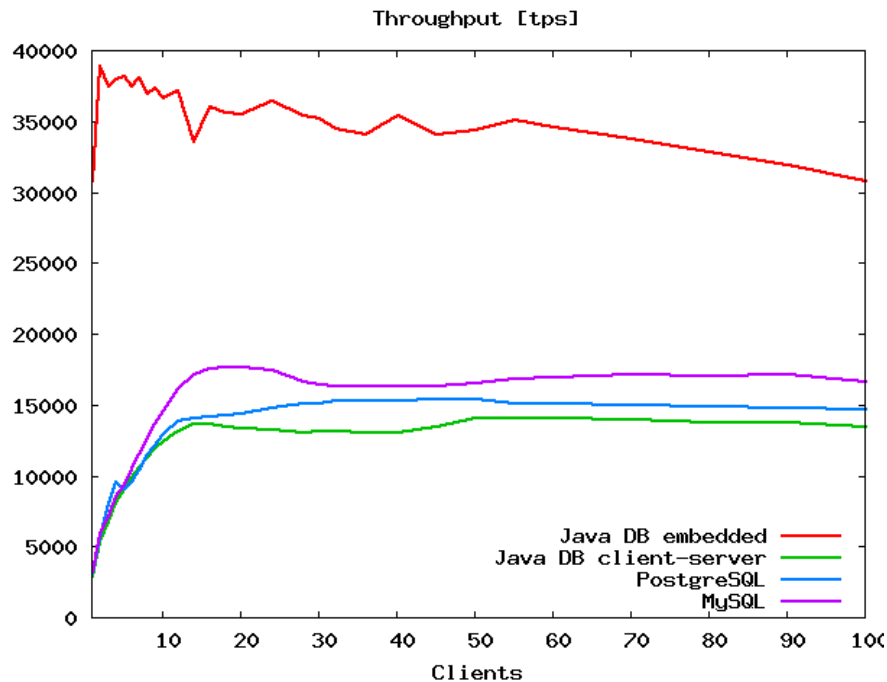
3 updates, 1 insert, 1 select

Test Configuration:

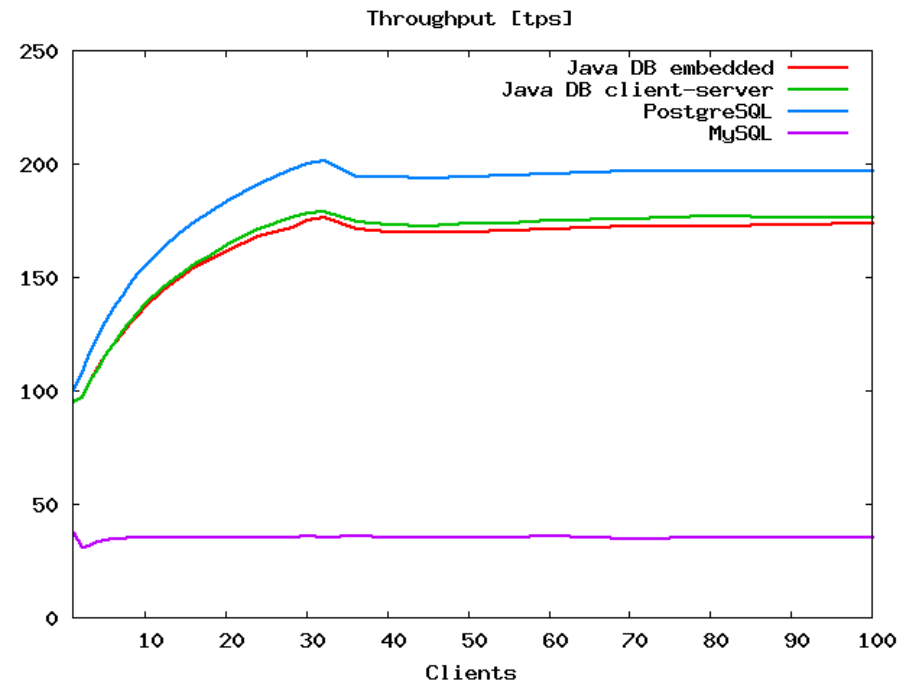
- > „Out-of-the-box“
- > 50 MB database buffer
- > Data and log on separate disks
- > SunFire v20z (2 AMD Opteron)
- > Solaris 10, Sun Java SE 6

Throughput: Single-record Select

Main-memory database (10 MB):

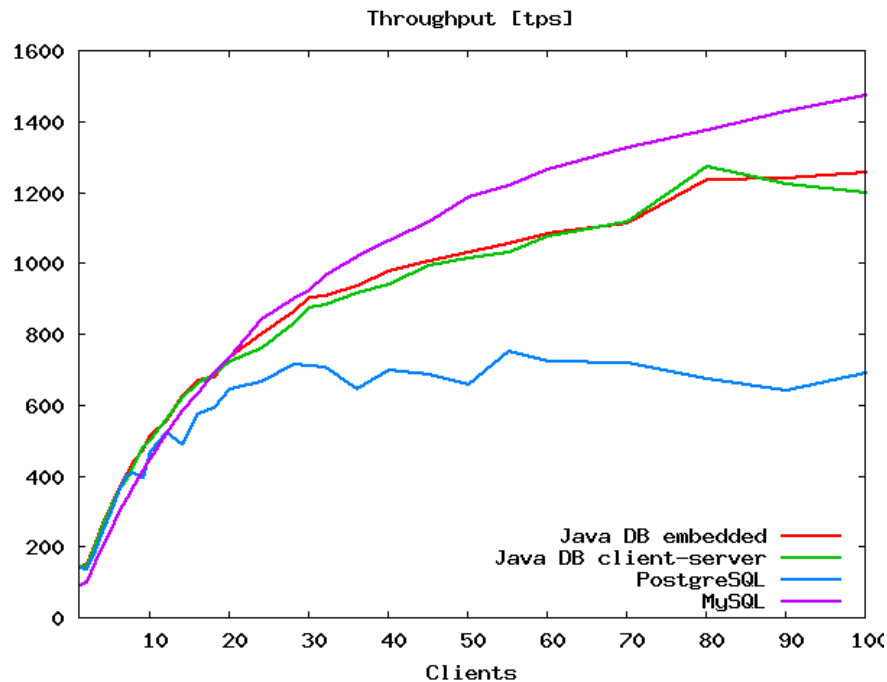


Disk-based database (10 GB):

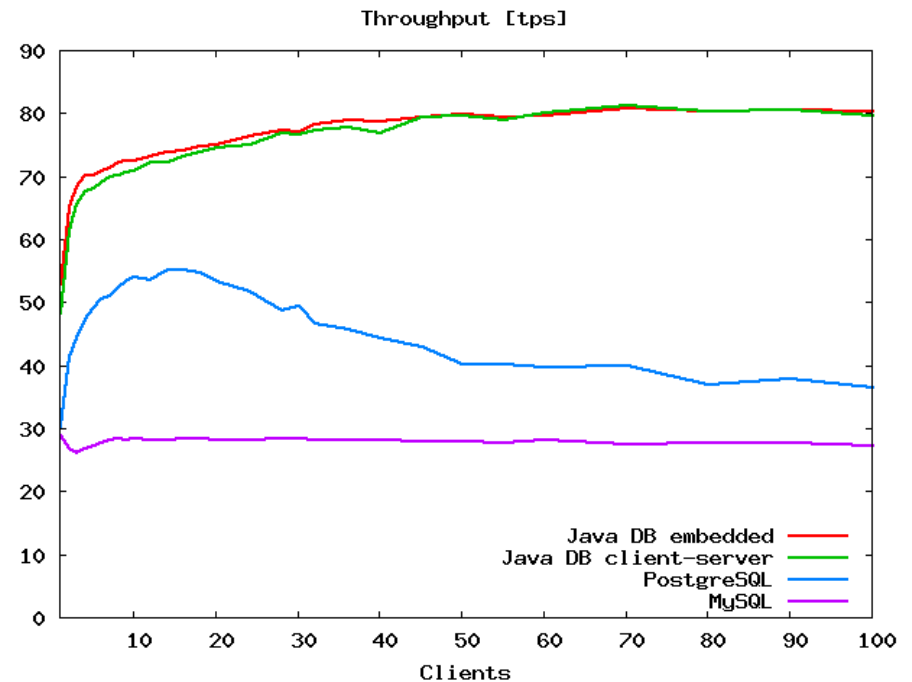


Throughput: Update Load

Main-memory database (10 MB):



Disk-based database (10 GB):



Summary

- > Trade-offs between durability and performance
 - know your requirements and select carefully
- > Know what influence performance
 - Java DB configuration
 - user application
- > Tips and tools to find and solve performance bottlenecks

Java DB **Performs!**

- Comparable to other open-source databases

Olav Sandstå
Sun Microsystems

<http://developers.sun.com/javadb/>
olav@sun.com