Internals of Derby

An Open Source Pure Java Relational Database Engine

Daniel John Debrunner
STSM – IBM Data Management
(1.1 2004/09/20)

Colorado Software Summit

Introduction

- After a quick overview of Derby, I will present some of the internals of the technology.
- Internal features will be covered at a high level, including some of the lessons learnt over the years.
- Please feel free to ask questions as we go.
- This is not a complete guide to the internals of Derby, for that read the source code!

Agenda

- Derby Overview
- System internals
- Language internals
- Store internals

Apache Derby

- IBM contributed the Cloudscape source code to the Apache Software Foundation as Derby
- Apache DB project sponsored Derby into incubation at Apache
- Derby now up and running at Apache
- Derby is an effort undergoing incubation at the Apache Software Foundation. Incubation is required of all newly accepted projects until a further review indicates that the infrastructure, communications, and decision-making process have stabilised in a manner consistent with other successful ASF projects. While incubation status is not necessarily a reflection of the completeness or stability of the code, it does indicate

Brief History

- 1996 – Cloudscape, Inc startup – Oakland, CA
- 1997 – JBMS 1.0
- Apr 1999 – Cloudscape 2.0
- Dec 1999 – Acquired by Informix Software
- June 2001 – Cloudscape 4.0
- July 2001 – Acquired by IBM
- Dec 2001 – IBM Cloudscape 5.0
- 2003 – IBM Cloudscape 5.1, 5.1FP1 & FP2
    - Significant IBM use as a component
- August 2004 – Open Sourced as Derby at

Derby Embedded Engine

- Pure Java
- Embedded Database
- Small Footprint
- Standards Based
- Complete Relational Database Engine

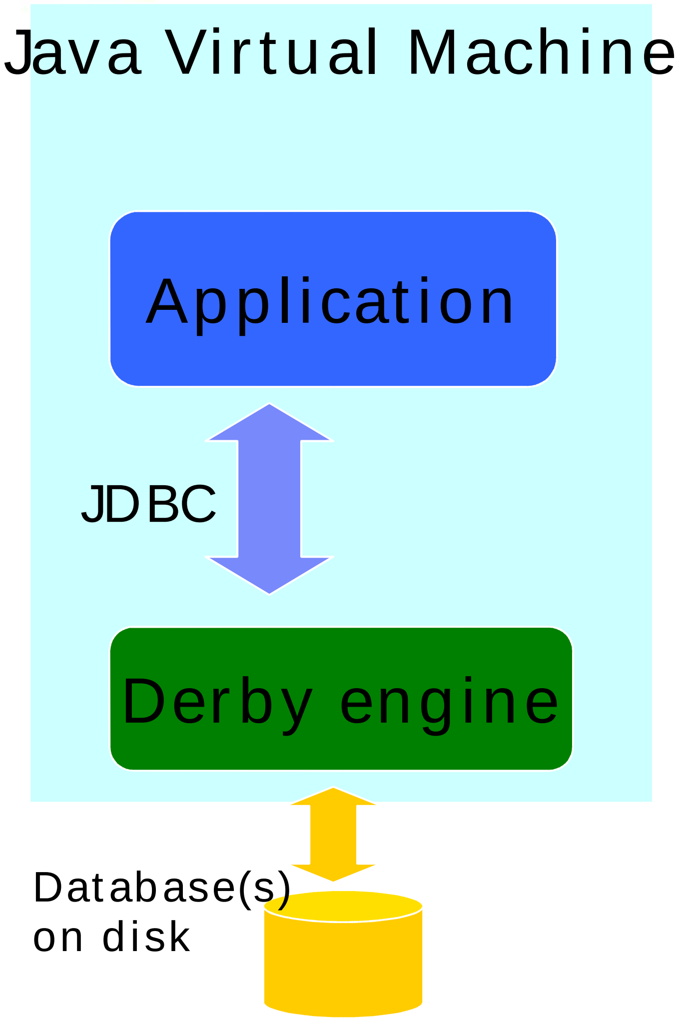
Pure Java

- Completely written in Java
- Supports J2SE 1.3/1.4/1.5 with single jar
- Runs anywhere – Linux, Windows, MacOS, Solaris, i-Series, z-Series
- Database format platform independent too

Embedded Database

- Database engine becomes integral part of the Java application
- No additional process
  - Application's JDBC calls now just result in method calls within same JVM
- Just a Jar file to the application
- Database invisible to end user of application

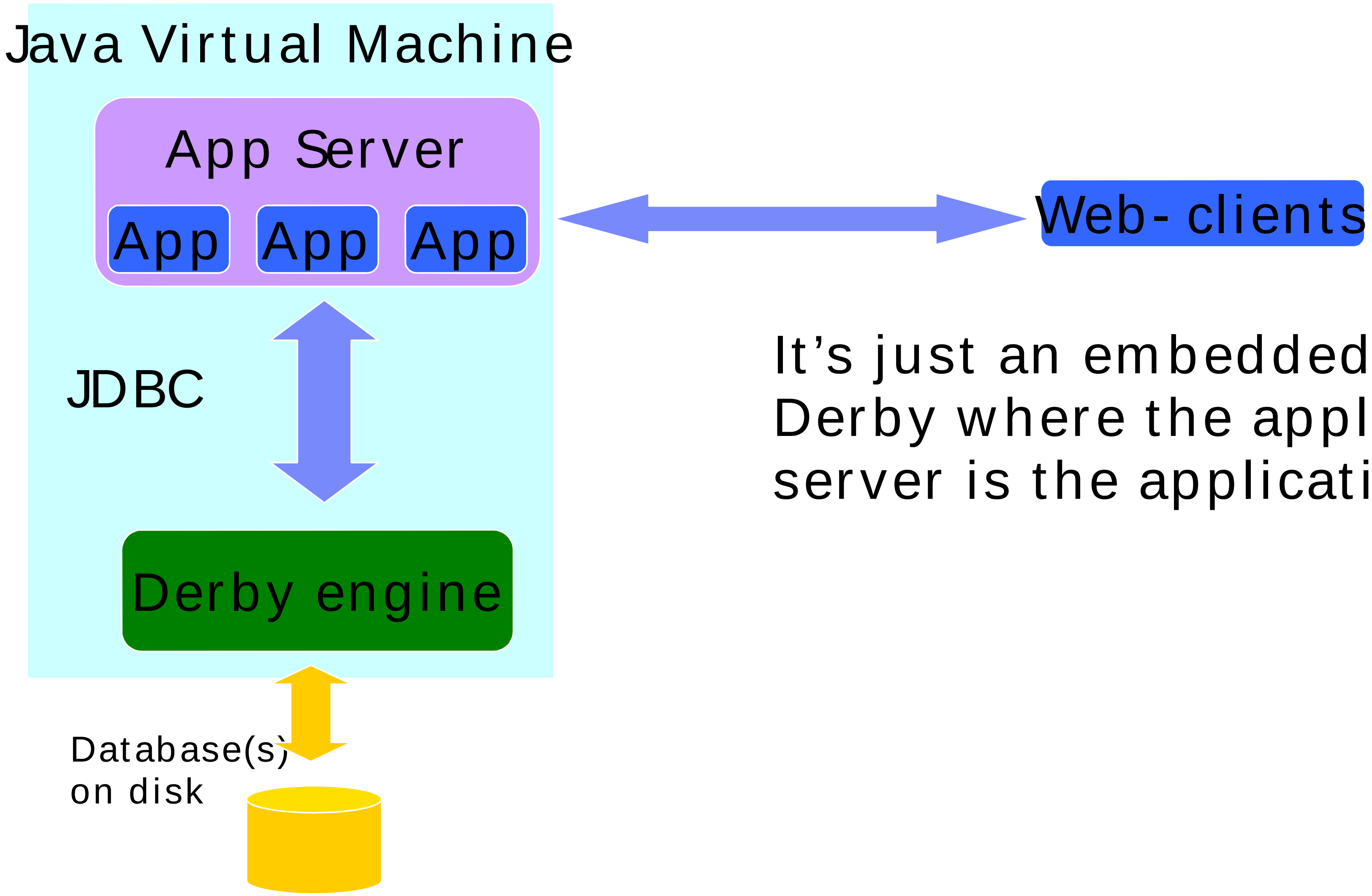
Embedded with Application



- Database only accessible from single JVM
- Java/ JDBC only
  No network connectivity

- Typically is single application per JVM (but could be multiple)

Embedded in Application Server



It's just an embedded use of Derby where the application server is the application

Small Footprint

- Engine jar file is around 2Mb
  - Optional Jar files
    - Network server ~150k
    - Tools ~200k
- Runtime memory use
  - Dependent on application, data caching, *etc*.
  - Can run when Java heap memory restricted to 4Mb
  - Have run in machines with only 16Mb physical memory

Standards

- SQL
  - SQL92, SQL99, SQL2003, SQL/XML, …
- Java
  - J2SE 1.3, 1.4
  - JDBC 2.0 & 3.0
  - J2EE – certified as a JDBC driver for J2EE 1.4 & 1.3
  - J2ME/OSGi
- DRDA

Complete Relational Engine

- Multi-user, multi-threaded, transactions, row locking, isolation levels, lock deadlock detections, crash recovery, backup & restore
- SQL
  - Tables, indexes, views, triggers, procedures, functions, temp tables
  - Foreign key and check constraints
  - Joins, cost based optimizer
- Data caching, statement caching, write ahead log, group commit
- Multiple databases per system

Agenda

- Derby Overview
- >> System internals
- Language internals
- Store internals

Original Cloudscape Intentions

- Database engine for highly pervasive market
- Small footprint for PDAs
- Written in Java for
  - Platform independence
  - Data independence
- Support multiple APIs
  - Possible low level storage API
  - Possible execute only engine

What Was Built

- JDBC driver that is a database engine
  - Trying to isolate JDBC from rest of engine increased footprint, needless conversions
- Footprint too big for PDAs
  - No PDAs running Java back in 1997
  - Ran on Psion 5mx with 16Mb, but boot time 40 seconds
  - Early customers wanted all the typical SQL features
- No low-level API, but much code infrastructure to support it
  - Adds code, multiple connection and transaction state objects

Typical JDBC Code

- ```
  PreparedStatement ps = conn.prepareStatement(
  “SELECT ORDERID, COST FROM ORDERS
              WHERE CID = ? AND COST > ?”);

  ps.setInt(1, customer);
  ps.setBigDecimail(2, threshold);

  ResultSet rs = ps.executeQuery();
  while (rs.next()) {
      int orderId = rs.getInt(1);
      BigDecimal cost = rs.getBigDecimal(2);
      // process order
  }

  rs.close();
  ps.close();
  ```
- Typical for web-applications, client applications will re-use PreparedStatement multiple times.
  ```

Typical JDBC Code

- ```
  PreparedStatement ps = conn.prepareStatement(
  "SELECT ORDERID, COST FROM ORDERS
              WHERE CID = ? AND COST > ?");

  ps.setInt(1, customer);
  ps.setBigDecimal(2, threshold);

  ResultSet rs = ps.executeQuery();
  while (rs.next()) {
      int orderId = rs.getInt(1);
      BigDecimal cost = rs.getBigDecimal(2);
      // process order
  }

  rs.close();
  ps.close();
  ```
- Every method call in red is a call into the JDBC driver
  ```

JDBC Threading Model

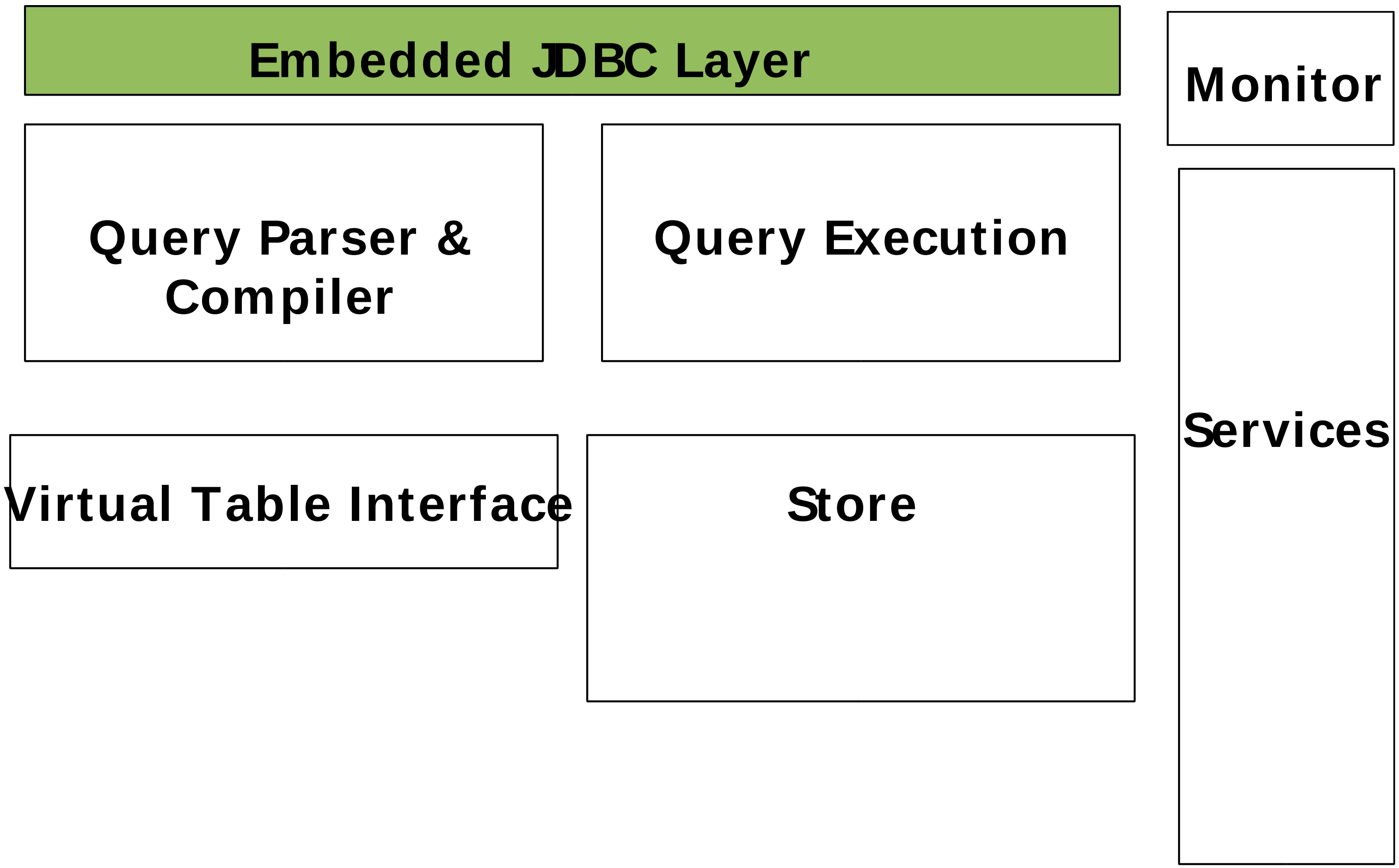
- ```
  (T1) conn.prepareStatement(
  “SELECT ORDERID, COST FROM ORDERS
              WHERE CID = ? AND COST > ?”)

  (T2) ps.setInt(1, customer)
  (T7) ps.setBigDecimal(2, threshold)

  (T4) ps.executeQuery()
  (T7) rs.next()
  (T1) rs.getInt(1)
  (T9) rs.getBigDecimal(2)


  (T2) rs.close()
  (T3) ps.close()
  ```
- Driver has to assume every call could be from a different thread
- 99.99% of the time, it won't be
  ```

Timing

- ```
  (T1) conn.prepareStatement(
  "SELECT ORDERID, COST FROM ORDERS
              WHERE CID = ? AND COST > ?")
  // could be significant time between prepare and execute

  (T2) ps.setInt(1, customer)
  (T7) ps.setBigDecimal(2, threshold)

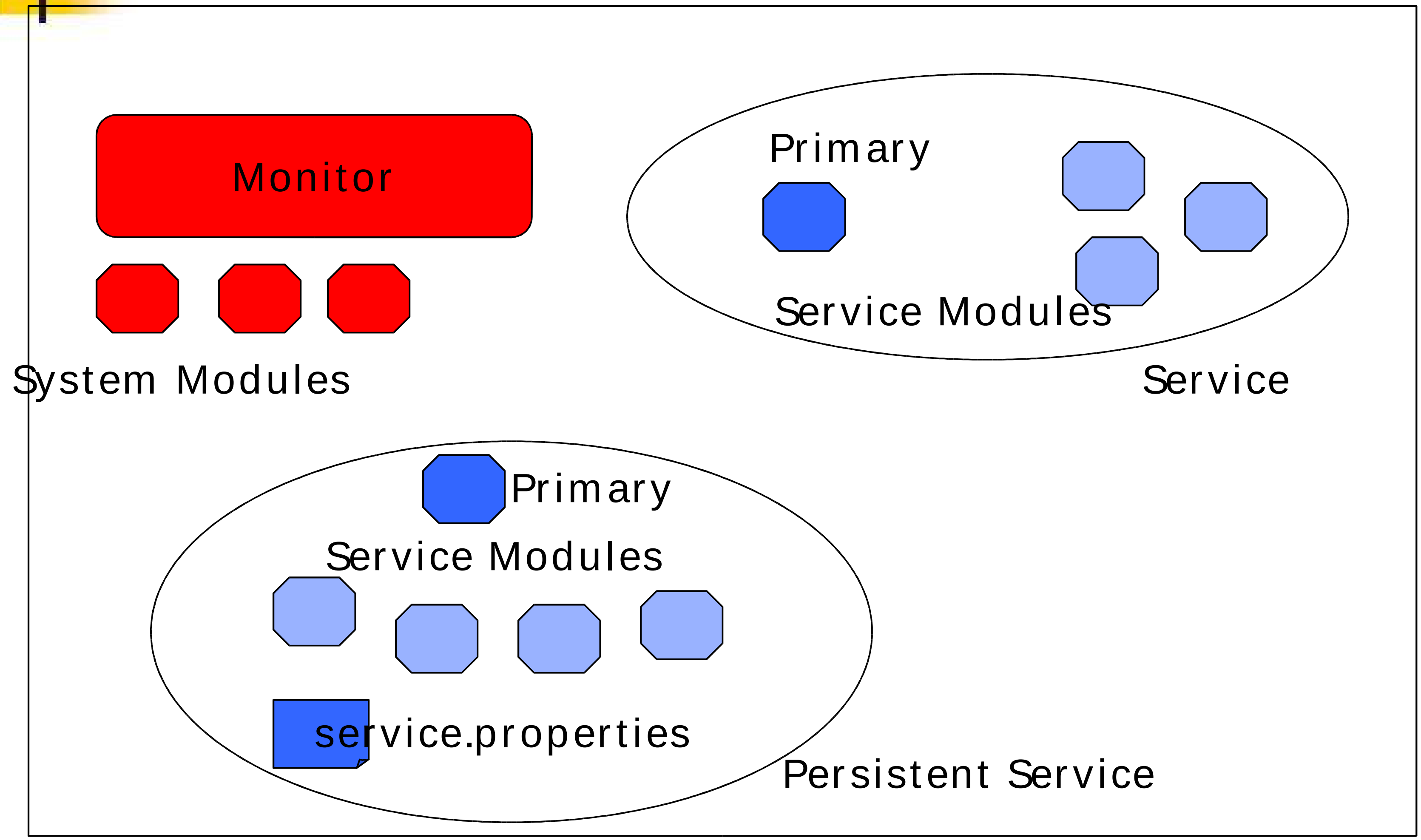
  (T4) ps.executeQuery()
  // could spend significant time in executeQuery
  (T7) rs.next()
  (T1) rs.getInt(1)
  (T9) rs.getBigDecimal(2)


  (T2) rs.close()
  (T3) ps.close()
  ```
- Two chances for underlying table to be modified, between prepare and execute, and in execute while waiting for intent locks on table
  ```

Logical Architecture



Module Architecture



Module

- Set of usable functionality
- Well defined API – “protocol”
- Protocol separated from implementation
- Typically declared as set of Java interfaces
- Identified by single Java interface, “factory class”.
  - *e.g.* org.apache.derby.iapi.services.locks.LockFactory
- May reference other module protocols
- Zero or more implementations in a running system
- Implementation can implement control interfaces to:
  - Define additional boot & shutdown actions
  - Define suitability for requested functionality

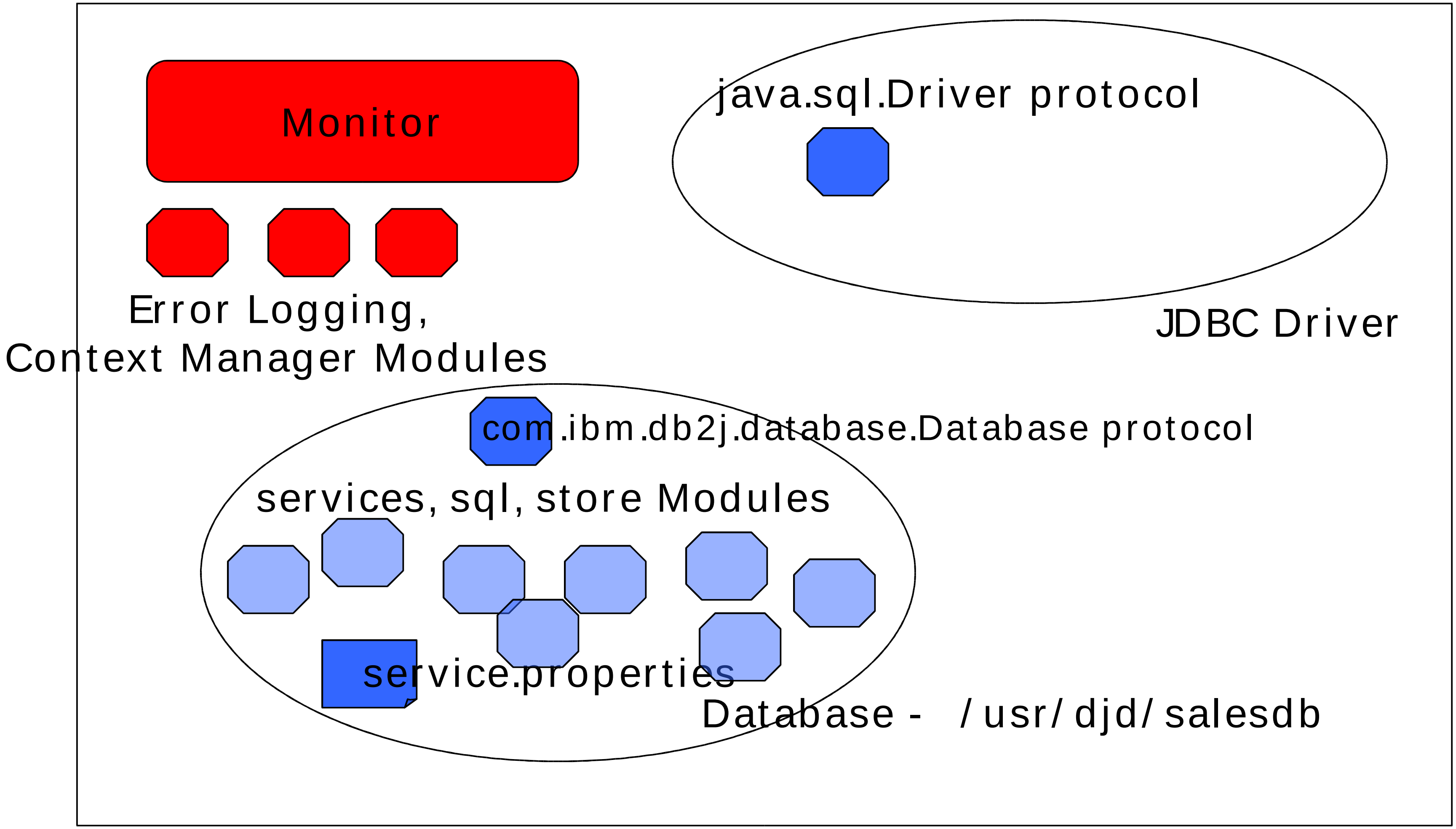
Service

- Collection of co-operating modules providing a complete set of functionality.
- Single primary module defining external API
- Persistent
  - Boot-up state in single service.properties file, including the required primary module identification (as the protocol, not the implementation)
- Non-persistent
  - Purely run-time definition.
- Modules always booted through monitor

Monitor

- Manages Derby system
- Boots & shutdown Services
- Finds service.properties file based upon service boot
- Maps requests for a module protocol to an implementation.
  Based upon:
  - Virtual machine environment (J2ME/J2SE 1.3/1.4)
  - Available classes (*e.g.* JCE encryption classes)
  - Suitability for current service
- Ensures system is not garbage collected away

Database Engine System



Why a Complex Monitor?

- Protocol separation from implementation
  - Good programming practice
  - Allowed early rapid development
  - Allows specific unit level module testing
- Single jar file supports multiple Java environments with no settings from application or user (ease of use)
  - JDK 1.1, J2ME, J2SE 1.2, 1.3, 1.4
- Supports different implementations from single source tree
  - Database, *Cloudsync server, Cloudsync target*
- Supports different store implementations with no change to language
  - Disk, read-only, database in a jar, database on HTTP server

Issues with Monitor

- Selection between implementations that satisfy requirements is not defined.
- Implementation list in single resource (modules.properties), no ability to add additional implementations in separate jar
- Allowing dynamic implementation selection could comprise security.

Session State & Error Cleanup

- Design goals
  - Avoid bugs in error handling code
  - Consistent resource cleanup
  - Modular state objects

Consistent Exception Clean Up

- Single exception model (StandardException)
- SQLException thrown through JDBC
  - Hence conversion always needed
- Five possible actions for an exception
  - Statement rolled back
  - Transaction rolled back
  - Session (connection) closed
  - Database close
  - System shutdown

Exception Model

- Originally five sub-classes, *e.g.* TransactionException
  - Complicates severity checking, adds code
- Now all action driven by integer severity level
- Exception cleanup for regular path as well. Connection.close() throws a close session severity exception
- Ensures cleanup code is correct, no hidden bugs due to infrequently executed cleanup

Context Cleanup

- Each session has a stack of context objects
- Each context object maintains session state and handles cleanup

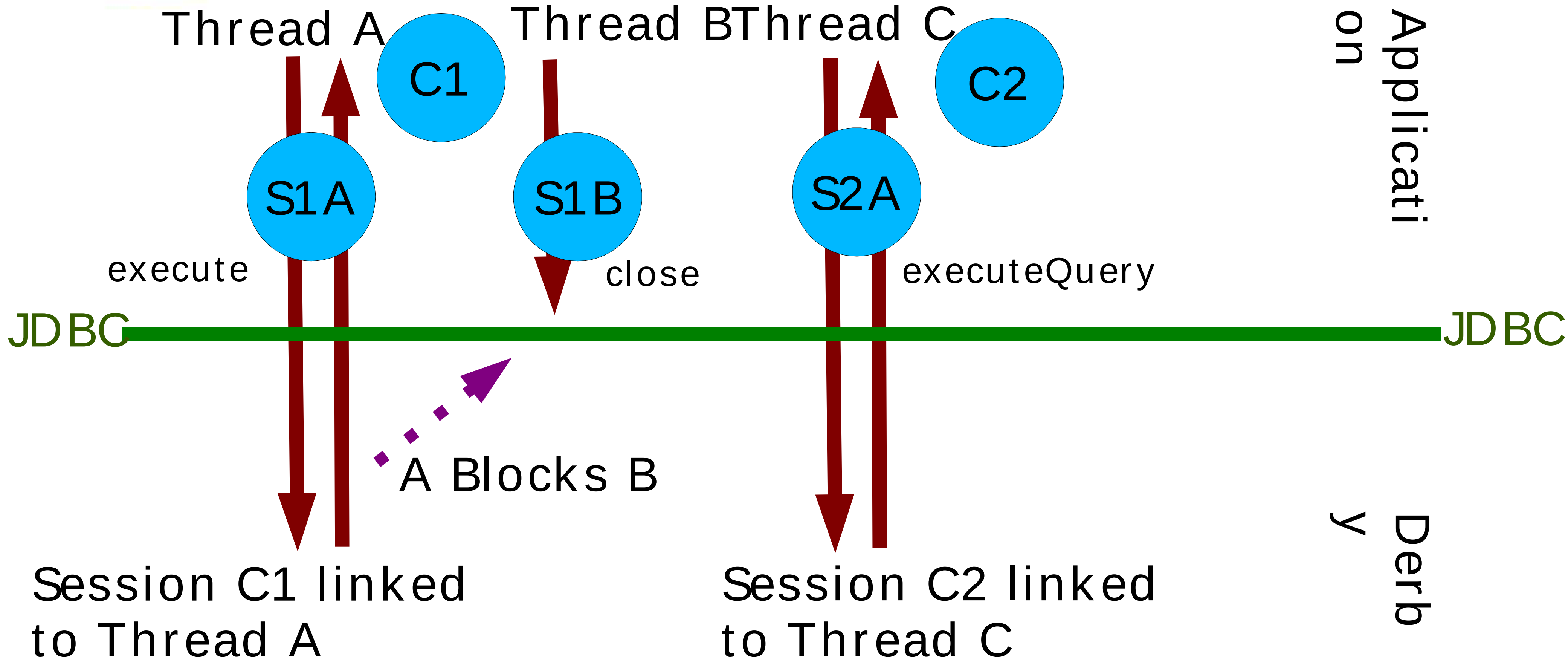
Database
JDBC Connection
LanguageConnection
Transaction
Statement
Statement
Compile

Each context is passed th
exception & performs
its own cleanup, includin
possibly popping itself o
the stack
Indicates if it is the last
context for that severity

Thread to Context Mapping

- Derby & JDBC do not hard link Thread to Connection
- Application may use any Thread to execute methods on any Connection or JDBC object
- Derby links application Thread to Session/Connection for the lifetime of the JDBC method call
  - All work performed using application's thread, not Derby worker threads

Single Active Thread per Session



While in Derby space Thread can find its Context Stack object with no state

Thread Mapping Implementation

- Originally used HashMap with Thread reference as key, stack as value
  - Slow, single threaded
- Now use Java 2 ThreadLocal
  - Faster, but still costly
  - Not supported on some J2ME configurations
- Optimised to avoid mapping when not strictly required
  - ResultSet.getXXX() methods, too expensive to map Thread to context every time

Improved Context Mapping

- Ensure Context stack is always available through method call parameters
  - Directly as passed parameter
  - Indirectly through fields or methods in passed parameters
- Large amount of code to re-work
- JDBC requires stateless mapping due to connection access in server side methods
  - DriverManager.getConnection ("jdbc:default:connection");

Thread Safe

- Single active thread per session simplifies state management within a session
- Derby code still multi-thread aware for shared resources such as data cache, lock manager, statement cache, *etc*.
- Thread safe but not optimised for multiple CPUs, hidden in implementations of caches and lock manager

Generic Cache/ Lock Managers

- Cache Manager caches objects that implement a Cacheable interface
- Handles aging, pruning, *etc.*
- Caches pages, statements, string translations, open files, dictionary objects
- LockManager locks objects that implement a Lockable interface
- Lock compatibility defined by Lockable, not manager
- Centralizes "hard" thread-safe issues

Agenda

- Derby Overview
- System internals
- >> Language internals
- Store internals

SQL Compilation

- PreparedStatement ps = conn.prepareStatement(“SELECT * FROM T WHERE ID = ?”);

1) Look up in cache using exact text match
(skip to 5 if plan found in cache)

2) Parse using JavaCC generated parser

3) Bind to dictionary, get types, *etc*.

4) Generate code for plan

5) Create instance of plan

Parse Phase

- Tree of Query Nodes created
- Many Nodes, one per operation type
  - FromBaseTable, MethodCallNode, DistinctNode, CurrenUserNode, ...
  - Bulk of code footprint is SQL compiler
- Switch to smaller number of building blocks?
- No execution code in Query Nodes, leads to similar set of execution ResultSets

- Duplicate checking of state in Query

Generate Phase

- Generate Java byte code directly, into in-memory byte array
- Load with special ClassLoader that loads from the byte array
- Single ClassLoader per generated class
  - Allows statements to be aged out independently
- Generated class extends an internal class BaseActivation which provides support methods, common functionality

Activation – Instance of plan

- Instance of generated class called Activation
- Holds query specific state, parameters, *etc*.
- Connected indirectly to JDBC PreparedStatement through holder/wrapper class that implements Activation interface
- Holder allows compiled plan to change without knowledge of application, transparent to PreparedStatement

Activation – Execution

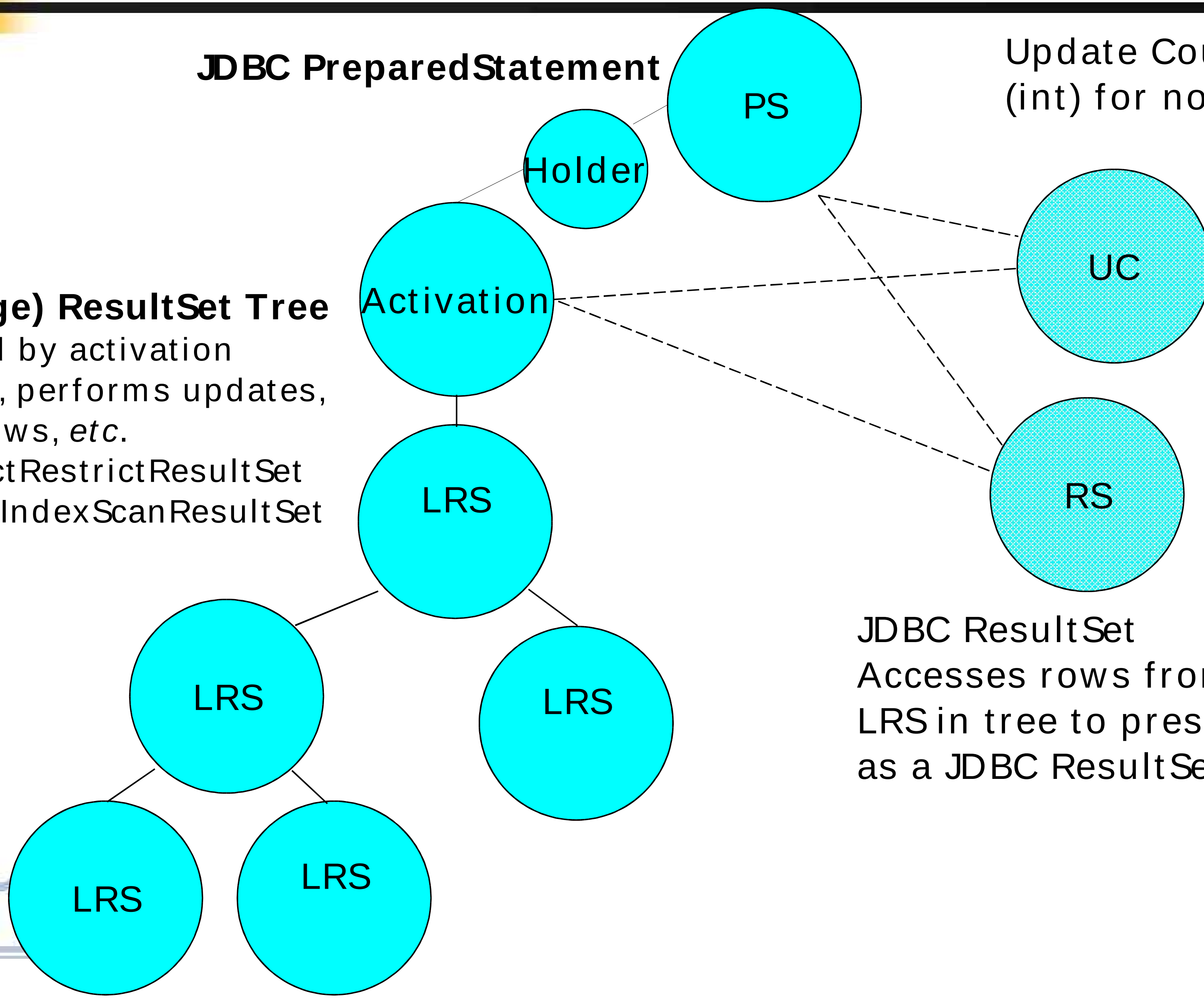
- PreparedStatement.execute methods
- Creates tree of internal ResultSet objects that map to SQL operations
  - ScanResultSet
  - SortResultSet
  - IndexScanResultSet
- Generated code main glue code
- Expressions are generated as methods in generated class

Statement Results

JDBC PreparedStatement

Update Count
(int) for non-queries

(Language) ResultSet Tree
Generated by activation execution, performs updates, fetches rows, *etc*.
e.g. ProjectRestrictResultSet on top of IndexScanResultSet



JDBC ResultSet
Accesses rows from top LRS in tree to present them as a JDBC ResultSet

Benefits of Generated Code

- No need to have Derby specific interpreter written in Java, just use JVM
- Generated code will get JIT'ed and thus gain the performance benefits
- Tight integration with Java calls from SQL, *e.g.* SQL functions written in Java. No use of (slow) reflection, just compile method call into generated class

Issues with Generated Code (1)

- First run on a new Java Virtual Machine pretty much guaranteed to break, find VM bugs
  - Derby generates code to JVM specification
  - VMs tended to only expect code as generated by Java compilers
  - Much better in recent years
  - Sun and IBM use Cloudscape database tests in their nightly VM testing

Issues with Generated Code (2)

- Debugging – hard, no source, no byte code file, no line numbers, no mapping to elements in SQL statement
  - Debug options to:
    - dump byte code file
    - add “line numbers” that map to byte code offsets
- In system with multiple active statements can be hard to figure out which generated class maps to which SQL statement

Generator History

- Generator module initially used Java source code & javac for speed of implementation
- Then direct byte code implementation, using same APIs (interfaces)
- Byte code used for product, java source for debugging
- Compilation was slow, significant time spent in generation

ByteCode Generator Issues

- Too many objects created – object per byte code instruction
- Too many classes – footprint issues
- Tree of objects representing generated class mimiced structure of class format generator
- API was not neutral, translated naturally into Java source, not into byte code

ByteCode Improvements

- Changed API to match fast byte code generation – dropped Java source implementation
- QueryNodes generate almost at the byte code level, definite knowledge of stack based VM
- Number of classes reduced to 16 (4,12) from 65 (14,51)
- Close integration with class format builder
- Method code arrays built as-you-go

Generator Optimizations

- Methods added to activation interface for 10 expressions, allows direct execution rather than through reflection
  - Still support arbitary number of expressions, others will be called through reflection
- Generated second class to act as an Activation factory, avoids use of reflection to create instance of generated class
  - Not used as most JVMs now optimize

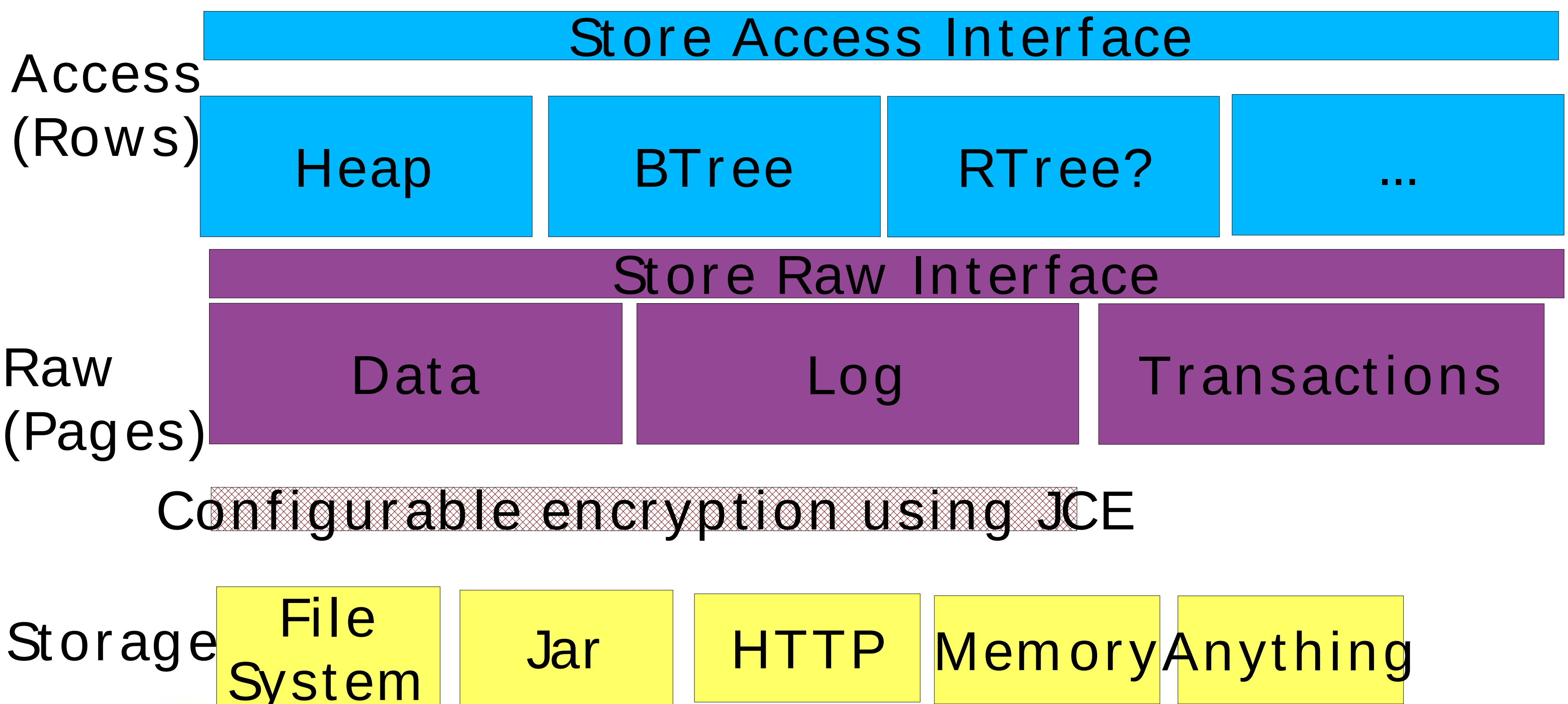
Artifact of Compile Model

- Each SQL statement involves class generation and loading, performance hurt by multiple statements like
  - INSERT INTO T VALUES(1, 'fred');
  - INSERT INTO T VALUES(2, 'fred');
  - INSERT INTO T VALUES(3, 'nancy');
- Correct approach is PreparedStatements
  - INSERT INTO T VALUES(?, ?);
  - Standard recommended JDBC practice, will perform better on all databases, but early JDBC programmers still use separate statements.

Agenda

- Derby Overview
- System internals
- Language internals
- >> Store internals

Store Architecture



Store Notes

- Store unaware of SQL semantics
- Index to heap linkage controlled by language layer, not store
- All language access through rows in a container, no concept of pages
- Fixed row model – enables row level locking

Flexible Store

- Designed to store serialized Java objects
- Variable length with no maximum defined
- Hence no limits on column size or row size
- Side effect of ease of use
  - Row always fits, regardless of page size
- Also considered:
  - Java class pre-processing
  - Shadow Java classes for storage

Flexibility Downside

- Column field lengths stored in each value
- No optimization for fixed length fields
  - E.g. SMALLINT value is prepended by two status bytes
    - Field state (NULL bit)
    - Field length (always 2) - “compressed integer”
- No fast access to N^{th} field, need to walk through previous fields

Transaction Logging

- Aries logging system for rollback and recovery
- Order of a change – write ahead logging
  - Write log record
  - Modify data page
  - On commit flush log up to point containing commit record
  - On data page flush, ensure log records for all modifications are flushed to disk

Log Record

- On recovery initially the 'do' action of a log record is made
- Subsequently the 'undo' action may occur if transaction was not committed before crash
- Log Records are objects in Derby
- 'do' method used for runtime application as well as recovery
- 'undo' method used for runtime rollback and recovery
- Single code path, no recovery specific

Resources

- Apache site – www.apache.org
- Derby site – incubator.apache.org/derby
- IBM Cloudscape – www.ibm.com/developerworks/cloudscape
- JDBC – java.sun.com/jdbc

- Dan Debrunner
  - debrunne@us.ibm.com (IBM)
  - djd@debrunners.com (Apache Derby)