Batch System Design

Snehal S. Antani
antani@us.ibm.com
Business Strategy Lead,
WebSphere Cloud & Virtualization Technologies
http://snehalantani.googlepages.com
Agenda

- Batch Architectures
  - Sequential Data Processing
  - Event/Random Data Processing
  - Unstructured/Semi-structured and super-scale data processing

- Application Design Goals
  - Agile Applications to improve time to market
  - Ensure High Performance

- Agile Applications
  - Foundational patterns to use when building applications
  - Shifting the burden of parallel processing to the infrastructure
  - Designing Shared Services
  - An example application architecture

- High Performance Applications
  - Managing referential data
  - Parallelizing validations, readers, processors, and writers
  - Performance analysis and identifying performance regressions

- Development Approaches
  - Domain-driven development and business-driven assembly
  - Generating service wrappers
  - Deployment Strategies

- Conclusions
Goal

Discuss how to build **agile, high performance** batch solutions

**Agile:** Share business services across batch & OLTP; Improve time to market through function reuse; leveraging container-managed services & frameworks

**High Performance:** Exploit parallel processing; leverage bulk IO optimizations; continuous performance regression testing;
Types of Batch Architectures

Random/Event-driven Data Processing
- Producer/Consumer
- Message-Driven Beans

Sequential Data Processing
- DataStream Driven
- WebSphere Compute Grid

Unstructured/Semi-Structured or super-scale (>500TB) Data Processing
- Map/Reduce
- Hadoop
Queue-based batch processing

1. Asynchronously process 1 record at a time
   \((X\text{ records} = X\text{ transactions}, X\text{ select statements}, X\text{ RPC's for insert statements})\)

2. **Used when Random access to data is required**

3. Solved using message-driven beans and JMS

4. Resource consumption managed by manipulating number of MDB threads

5. Parallelism provided by running multiple MDB’s across a cluster of servers

6. Limited options for operational control
Sequential Data Processing

1. Asynchronously process chunks of records at a time
   \((X \text{ records} = X/M \text{ transactions}, 1 \text{ select}, X/N \text{ RPC’s for inserts})\)

2. Should be used for **sequential data access**

3. Execute a **single select against the database** using holdable cursors

4. Leverage **JDBC Batching for writes**

5. Periodically commit M records in a given Checkpoint Interval (aka Syncpoint)

6. Bulk Record Processing **significantly reduces infrastructure overhead!**
Unstructured/Semi-Structured and Super-scale data processing

“Get assets for portfolio X”

**Map:** execute query in each worker  
**Reduce:** aggregate and sort the assets

Composed of 2 components: *master* and multiple *workers*

- **Map**  
  - the *master* breaks a large task up into many smaller tasks  
  - execute smaller tasks across a cluster of *workers*  
  - workers return its answer to the master.

- **Reduce**  
  - the *master* aggregates (and filters) the answers returned from each of the workers  
  - master returns aggregated results to the caller
Combining Sequential + Super-scale Data Processing - Example

Batch loop with checkpoint/restart

Portfolio DB → Batch Data Stream → Calculate Risk For Portfolio → Batch Data Stream → Portfolio DB

"Get assets for portfolio X"

For each portfolio in the DB {
  get list of all assets;
  compute portfolio risk;
  store new portfolio risk;
}

- Provides restartability when processing millions of portfolios
- Leverage map/reduce to search through GB/TB of asset data
- Use Compute Grid PJM to parallel-process portfolios, partitioning on a portfolio-basis
Approaches for High Performance Batch

• *Proximity to the Data*
  – Bring the business logic to the data: co-locate on the same platform
  – Bring the data to the business logic: in-memory databases, caching

• *Data-aware Routing*
  – Partitioned data with intelligent routing of work

• *Divide and Conquer*
  – Highly parallel execution of workloads across the infrastructure

• *On-Demand Scalability*
Proximity of Data - Options

1. WAS z/OS using optimized mem-to-mem JDBC Type-2 Driver
2. WAS z/Linux using JDBC Type-4 driver and SSL over optimized z network stack
3. WAS distributed (unix/linux/windows/etc) using JDBC Type-4 driver and SSL over traditional network stack
4. WAS distributed coupled with WebSphere eXtreme Scale cache
Compute Grid + XTP = eXtreme Batch

*Bringing the data closer to the business logic*

- Proximity of the business logic to the data significantly influences performance
  - Bring data to the business logic via caching
  - Bring business logic to the data via co-location

- Increase cache hits and reduce data access through affinity routing
  - Data is partitioned across the cluster of workers
  - Work requests are divided into partitions that correspond to the data
  - Work partitions are intelligently routed to the correct work with the data preloaded.
Understanding Data Access

Data Access time (ms) =
(Probability of near-cache hit) \times (Time to retrieve data from near-cache) +
(Probability of near-cache miss) \times (Time to retrieve data from other storage);

Time to retrieve data from other storage (ms) =
(Probability that data is in cache server) \times (Time to retrieve data from cache server) +
(Probability that data must be retrieved from database) \times (Time to retrieve data from database);
Example calculation

\[
\text{Data Access} = (\text{Near-Cache Hit}) + (\text{Near-Cache Miss})
\]

\[
\text{Near-Cache Hit} = (P_1)(S_1)
\]

\[
\text{Near-Cache Miss} = (P_2) \times [(P_3)(S_3) + (P_4)(S_4)]
\]

Near-cache miss = \((.7)(10) + (.3)(200)\)

\[= 7 + 60 = 67 \text{ ms}\]

\[
\text{Data Access} = (.3)(1) + (.7)(67)
\]

\[= .3 + 46.9 = 47.2 \text{ ms}\]
Example calculation - effects of applying data-aware routing.

\[
\text{Data Access} = (\text{Near-Cache Hit}) + (\text{Near-Cache Miss})
\]

\[
\text{Near-Cache Hit} = (P_1)(S_1)
\]

\[
\text{Near-Cache Miss} = (P_2) \cdot [(P_3)(S_3) + (P_4)(S_4)]
\]

Near-cache miss = \(0.7 \times 10 + 0.3 \times 200\)
\[= 7 + 60 = 67 \text{ ms}\]

\[
\text{Data Access} = (0.6)(1) + (0.4)(67)
\]
\[= 0.6 + 26.8 = 27.4 \text{ ms}\]

\[\frac{(47.2 - 27.4)}{47.2} = 42\% \text{ improvement in data access time}\]
Developing Batch Applications
Batch Platform + OLTP (WLM Integration, Job Pacing, Job Throttling, ...)

Batch Platform (Parallel Processing, Job Mgmt, Scheduler Integration)

Batch Container (Checkpoint/Restart, Execution Metrics, ...)

EJB 3  EJB 2  Spring Batch  C.G. API  BDS Framework

BDS Framework + Spring  Pojo – Annotation – AOP

Data Fabric

JDBC  JPA  Hibernate  WXS  ibatis  Pure Query
Batch Platform

- Jobs are submitted to the system via an enterprise scheduler, process server, Job Dispatcher GUI, or programmatically via EJB, JMS, or Web Services
- Administrator can define dispatch & partitioning policies
- Job Dispatcher selects the best endpoint for job execution based on execution metrics
- Job Dispatcher aggregates job logs and provides life-cycle management (start/stop/cancel/etc)
Components of a Batch Application

- Where does the data come from?
- Execute Step N if Step N-1 rc = 0
- How should the Step be: - Check pointed? - Results processed? Etc…
- Where should the data be written to?
- How should the business logic process a record?

Start Batch Job

Step 1

… Step N

Complete Batch Job

Input BDS

Output BDS

Input BDS

Output BDS

Complete Batch Job
Foundational Patterns

**Strategy Pattern:** allows the isolated replacement of algorithms

Separation of functional concerns: compliments the strategy pattern, where functional boundaries are decoupled through interfaces
How to think about batch jobs

- Algorithms should be written to **operate on a single record:**

- Reader Algorithm is a domain object factory, producing a single record for processing
- Processor Algorithm operates on a single record: is unaware how the record was obtained
- Writer Algorithm maps a single record to bytes: expects a valid record that can be persisted

- Batch Container & Batch Step Wrapper are responsible for invoking each functional component
1. Batch Input Data Stream (Input DS) manages acquiring data and creating domain objects
2. Record processor applies business validations, transformations, and logic on the object
3. Batch Output Data Stream (Output DS) persists the output domain object
4. Processor and OutputDS are not dependent on Input method (file, db, etc)
5. Processor and OutputDS only operate on *discrete business records*
6. Customers can use favorite IOC container to assemble the Kernel, and use xJCL to wire the batch flow (input -> process -> output) together.
- Writing multi-threaded code is fun, but hard!
- Multi-threaded code can be difficult to debug, and expensive to maintain
- The degree of parallelism and where parallel jobs are executed should be a **point-in-time decision**, where the infrastructure and operations team(s) decide based on:
  - Impact on SLA’s & deadlines
  - Current data partitioning schemes
  - Available capacity of endpoints
- Algorithms should be written as **stateless, single-threaded** code that **processes 1 record**
Parallelism: An Infrastructure Problem

- Compute Grid Parallel Job Manager creates and dispatches multiple instances of the batch job
- Each instance operates on its own data segment using constrained SQL queries (for example)

```
select account ... from t1
where account > X and account < Y
```

```
insert balance into ...
```

```
Multi-threaded Validation
```

```
Cached Validation
```

```
Input Dom. Obj
```

```
Output Dom. Obj
```

```
Kernel
```

```
Job Instance 1
```

```
Job Instance 2
```

```
Job Instance 3
```

```
X=000 Y=299 Job Instance 1 Job Instance 2 Job Instance 3
```

```
Parallelism: An Infrastructure Problem
```

```
Compute Grid Parallel Job Manager
```

```
X=700 Y=999
```

```
X=300 Y=699
```

```
X=000 Y=299
```

```
Input DS
```

```
Output DS
```

```
Processor
```

```
Validation...
```

```
Validation N
```

```
Output Dom. Obj
```

```
Input Dom. Obj
```

```
select account ... from t1
where account > X and account < Y
```

```
insert balance into ...
```
Parallelizing Across the Infrastructure

1. Parallel Partitioning Policies define how the job should be broken up
   - Application developers write single-threaded pojo’s, infrastructure determines the degree of parallelization

2. The aggregated job output is streamed to the user.

3. Integrate with WebSphere eXtreme Scale or SolidDB for “eXtreme Batch”
eXtreme Transaction Processing with Compute Grid
Sharing Services Across Batch and OLTP

**Challenge**: share services across different execution styles, without sacrificing the performance optimizations each style offers

**Online Transaction Processing (OLTP)**: Random data access w/ conversational state. Data grid technologies (ie. WebSphere eXtreme Scale) to improve performance

**Batch Processing**: Sequential data access. Efficiencies when processing data in bulk

**Real-time (Queue-based) Processing**: Random data access. Data grid technologies & data-aware routing to improve performance

**Future?** Map/reduce, cascading, etc
Designing a Shared Service

Data Injection versus Data Acquisition

- Data Acquisition: The service is responsible for obtaining the record to process
- Data Injection: The service doesn’t know (nor care) how the record was obtained

- **Data Acquisition** is the typical approach in OLTP
- **Data Injection enables services to be shared** across multiple execution styles
- Using ORM wrappers (Hibernate, etc) recklessly **can limit the ability to share services**!

```java
Public AccountObject AccountReconciliationService(String key) {
    accountObject = DAL.getAccountObject(key);
    // code to reconcile account data
    return accountObject;
}
```

**Data Acquisition**

```java
Public AccountObject AccountReconciliationService(String key) {
    accountObject = DAL.getAccountObject(key);
    accountObject = reconciliationSharedService.processRecord(accountObject);
    return accountObject;
}
```

**Data Injection**
Designing a Shared Service

- Domain-specific wrappers are used to retrieve data
- Wrappers acquire data and pass it to the shared service

```java
public AccountObject AccountReconciliationService(String key) {
    accountObject = DAL.getAccountObject(key);
    accountObject = reconciliationSharedService.processRecord(accountObject);
    return accountObject;
}
```

OLTP

```java
public void AccountReconciliationBatchService(String key) {
    while (batchReader.hasNext()) {
        accountObject = batchReader.next(); // batchReader optimized for bulk reading
        accountObject = reconciliationSharedService.processRecord(accountObject);
        batchWriter.writeAccount(accountObject); // batchWriter optimized for bulk writing
    }
}
```

Batch

```java
public void onMessage(Message msg) {
    accountKey = getAccountKey(msg);
    accountObject = DAL.getAccountObject(accountKey);
    accountObject = reconciliationSharedService.processRecord(accountObject);
    return accountObject;
}
```

JMS
Sharing Services Across Batch and OLTP - The role of Containers

- Execution Containers (EJB Container, Batch Container) are responsible for managing transactions, security, etc.
Development Approaches
A Batch Application Development Approach (1)

Design Patterns, Domain-Driven Development, Business-Driven Assembly

Developers

BDSFW Steps

Shared Libraries

Pojo Input Streams
(Dom. Obj. Factories)

Pojo Business Logic
(Dom. Obj. Processors)

Pojo Output Streams
(Dom. Obj. Persisters)

xJCL Template

Rules Validation & Transformation Tools

Business Analysts

XML-based App Development
A Batch Application Approach (2)

- Application developers are exposed to Pojo’s and the DI/AOP Container
- OLTP and Batch Wrappers can be generated
- DBA’s can focus on writing highly-tuned SQL statements for bulk reading/writing
Conclusions
Business Benefit

… Cut IT development, operations, and maintenance costs by pursuing the “Unified Batch Architecture” strategy with Compute Grid

- Today: Batch processing systems exist in silos

![Diagram showing current and future batch processing system architectures](image)
The Batch Vision

1. Batch Containers should run everywhere
2. **Portable Batch applications** across platforms and J2EE vendors
3. Location of the data dictates the placement of the batch application
4. Centrally managed by your enterprise scheduler
5. Integrating with existing: Disaster Recovery, Auditing, Logging, Archiving
References

- WebSphere Extended Deployment Compute Grid ideal for handling mission-critical batch workloads

- CCR2 article on SwissRe and Compute Grid

- WebCasts and Podcasts on WebSphere XD Compute Grid
  http://snehalantani.googlepages.com/recordinginterviews

- Java Batch Programming with XD Compute Grid

- WebSphere Compute Grid Frequently Asked Questions

- Development Tooling Summary for XD Compute Grid

- Compute Grid Discussion forum

- Compute Grid Trial Download

- Compute Grid Wiki (product documentation)
Backup
WebSphere XD Compute Grid Summary

• IBM WebSphere XD Compute Grid delivers a complete batch platform
  – End-to-end Application Development tools
  – Application Container with Batch QoS (checkpoint/restart/etc)
  – Features for Parallel Processing, Job Management, Disaster Recovery, High Availability
  – Scalable, secure runtime infrastructure that integrates with WebSphere Virtual Enterprise and WLM on z/OS
  – Designed to integrate with existing batch assets (Tivoli Workload Scheduler, etc)
  – Supports all platforms that run WebSphere, including z/OS.
  – Experienced Services and Technical Sales resources available to bring the customer to production

• Is ready for “prime time”. Several customers in production on Distributed and z/OS today
  1. Swiss Reinsurance, Public Reference, Production 4/2008 on z/OS
  2. German Auto Insurer, Production 7/2008 on Distributed
  3. Turkish Bank, Production on Distributed
  4. Japanese Bank, Production on Distributed
  5. Danish Bank, Pre-production on z/OS
  6. Wall Street Bank (two different projects, globally deployed), Production on Distributed
  7. South African Bank, Pre-production on Distributed
  8. Danish business partner selling a core-banking solution built on Compute Grid.
  9. German Life Insurance Company. Pre-production on z/OS
  10. Spanish Bank. Pre-production on Distributed.
  – Numerous other customers in pre-production & PoC

• Vibrant Customer Community
  – Annual customer conference held in Zurich, hosted by Swiss Re.
  – User group established for sharing best practices and collecting product requirements
  – Over 800,000 reads in the Compute Grid developers forum since January 22nd, 2008. (~5k reads per week)
Developing High Performance Batch Applications
Dealing with Validations, Rules, Referential Data

- Validations often utilize referential data and/or business rules to evaluate a record
- Validations can significantly impact performance
  - if 1 validation takes 250 ms, then 1,000,000 records could take ~70 hours to process!

1. Can the validation be removed by adding a predicate to the SQL query in the bulk reader?
2. Can we cache some of the referential data?
3. Can we cache all referential data using a data grid (WebSphere eXtreme Scale)?
4. Can we improve the probability of a cache hit using data-aware routing or larger JVM’s?
Parallelizing validations, readers, processors, and writers

- 1 job runs on 1 thread in Compute Grid

- Discrete tasks can be executed in parallel
  - Next N records are fetched on a background thread
  - Validations are executed in parallel

- Discrete records can be processed concurrently

- Optimizations can be applied transparently to the application code (if you listen to us)!
Identifying Performance Regressions Early

- Inefficiencies in your code are magnified in batch processing!
- Bottom-up performance analysis & performance regression tests are important in batch!
- Simple tools and methodologies can be used to identify performance regressions

Test 1
Input Data → Reader Algorithm v1 → "Echo" Processor → System.out Writer → Output Data → Elapsed time X

Test 2
Input Data → Reader Algorithm v2 → "Echo" Processor → System.out Writer → Output Data → Elapsed time Y

- Input Data, Echo Processor, and System.out Writer are constant across tests
- Reader Algorithm v2 is tested for a performance regression
- If Y is z% more than X, there is possibly a performance regression in the reader
- Larger the input data, more accurate the elapsed time becomes.

- By holding various components constant, we can identify potential performance regressions in any of the algorithms, validations, etc
Identifying Bottlenecks at Run Time

- Functional separation of input - process - output allows us to identify bottlenecks at runtime
- Taking measurements before and after each functional call can identify where to focus

```java
for each record {
    startTime = System.getCurrentTimeMillis();
    record = readRecord();
    endTime = System.getCurrentTimeMillis();
    recordCount++;
    totalReadTime = totalReadTime + (endTime – startTime);
}
```
```
startTime = System.getCurrentTimeMillis();
processedRecord = processRecord(record);
endTime = System.getCurrentTimeMillis();
totalProcessTime = totalProcessTime + (endTime – startTime);
```
```
startTime = System.getCurrentTimeMillis();
writeRecord(processedRecord);
endTime = System.getCurrentTimeMillis();
totalWriteTime = totalWriteTime + (endTime – startTime);
```

// print average & total read/process/write times.

- The larger the input data, the more accurate the measurements become
- Finer-grained time calculations can be enabled within the input, process, and output components
An Approach to Evaluating Performance

1. Determine the time to process 1 record
   - Cost of the Reader algorithm
   - Cost of the processor (and its validations) algorithm
   - Cost of the Writer algorithm

2. Calculate the theoretical elapsed time for the job

3. Calculate the actual elapsed time for the job
   - Run the job sequentially in Compute Grid
   - Compare actual elapsed time to theoretical elapsed time
   - Vary checkpoint frequency to understand its impacts

4. Prove efficient parallel processing
   - Run N parallel jobs
   - Compare elapsed time to theoretical time / N
   - Compare elapsed time to actual sequential time / N

5. Stress & Tune a single JVM
Deployment Strategies

- 1 ear per batch step?

- 1 ear per “domain” (related business function, owning department, etc), shared libraries for BDSFW implementation classes

- Only 1 ear, shared libraries for BDSFW implementation classes

- Need to consider lifecycle management issues (versioning, continuous availability, version affinity, etc)
Execution Agnostic Application “Kernel”
Compute Grid (Batch) Wrapper to the Shared Service

Additional I/O types for batch can be added as new pattern implementations. For example, we can read from an NIS Dataset (r/O) by implementing the ZFileRecordOrientedReader and Writer. We can read/write from WebSphere eXtreme Scale using the DataGridReader/Writer, etc.
OLTP Wrapper to the Shared Service

Many Persistence technologies could be used from OLTP, without fear of degrading the performance of the application in "tech" mode, because we have the option of isolation (per this example).
The BDS Framework - Implements checkpoint/restart for common IO types

Batch Container

GenericXDBatchStep

FileReaderPattern

FileWriterPattern

ByteReaderPatternAdapter

ByteWriterPatternAdapter

The BDS Framework - Implements checkpoint/restart for common IO types
Batch Data Stream Framework
XD Compute Grid makes it easy for developers to encapsulate input/output data streams using POJOs that optionally support checkpoint/restart semantics.

Job Start

1. open()
2. positionAtInitialCheckpoint()
3. externalizeCheckpoint()
4. close()

Job Restart

1. open()
2. internalizeCheckpoint()
3. positionAtCurrentCheckpoint()
4. externalizeCheckpoint()
5. close()
Batch Data Stream Framework (BDSFW)

- Recognized that most of the Batch Data Stream implementations were common across customer engagements
  - ie. How many different ways can you really checkpoint a file?

- Implemented a Template/Callback framework
  - **Templates**: Implement as much of the Batch Data Stream interface as possible
  - **Callbacks**: Interfaces that complete the templates

- Best practices can be hardened in the Templates

positionAtInitialCheckpoint() {
    // get connection
    // get SQL query
    // create prepared statement
}

BDSFW JDBCReader Template

AccountReader implements JDBCReaderPattern {
    //....
    public String getSQLQuery() {
        return "select accounts from accountTable";
    }
    // ...

JDBCReaderPattern Callback
Batch Data Stream Framework (BDSFW)

<bds>
   <logical-name>inputStream</logical-name>
   <props>
      <prop name="ds_jndi_name" value="${accounts.jndiname}"/>
      <prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.accounts.AccountReader"/>
   </props>
   <impl-class>
      com.ibm.websphere.batch.devframework.datastreams.patterns.LocalJDBCReader
   </impl-class>
</bds>

- xJCL snippet that wires together the Template (LocalJDBCReader) and the Callback (AccountReader)
Batch Datastream Framework

- Selector pattern, applied via xJCL, for choosing which input and output BDS template to use

- GenericXDBatch Step implements the various patterns described, including:
  - Performance metrics (coarse-grained & fine-grained) for functional components
  - Strategy pattern enabling algorithms & call-backs to be plugged in
  - Threshold policies, transactional error streams, etc

- **Futures**: parallel readers/processors/writers can be built into GenericXDBatchStep, and transparently applied to customer code.
End-to-end Development tooling

- Customer develops business service POJO’s
- Applications are assembled via IOC Container
- XD BDS Framework acts as bridge between job business logic and XD Compute Grid programming model
- XD Batch Simulator for development
- XD Batch Unit test environment for unit testing
- XD batch packager for .ear creation

Java IDE

Business Services

Compute Grid Pojo-based App

CG BDS Framework

Eclipse-based CG Batch Simulator

Single Server CG Batch Unit & Function Test Environment

CG Batch Packager

Business Services Testing Infrastructure

Single Server Unit-testing for OLTP

Common Deployment Process

Compute Grid Infrastructure