JSimpleDB: Language Driven Persistence for Java

Archie Cobbs

Emergency Callworks, Inc.
aka Motorola Solutions

Why Persistence?

- Computers are state machines and persistence allows flexibility in handling that state
- Machines crash, power cycle, reboot, ...
- Code is always evolving
- Program state may not all fit into memory
- Multiple computers accessing the same state
- Separation of concerns: code vs. data
- **Persistence is part of almost all software**

Programming Language Fantasyland

- Data access is atomic
- Data access is instantaneous
- Data access never fails
- Data has has no encoding issues
  - Data can never be corrupted
- Data always behaves the same way
- No other computer can touch my data

The Persistence Reality

- All data must be encoded into 1’s and 0’s
- Encodings can change between code versions
- Data access can be slow
- Data access can fail randomly
- Data access is not atomic
  - Transaction semantics can be very subtle
- Other machines can touch my data

The Problem

- Persistence is basic to real-world programming
- Yet programming language designers have never addressed persistence
- Result: persistence technologies have been shoehorned into programming languages
- Examples:
  - Shoehorn #1: JDBC
  - Shoehorn #2: JPA, JDO
- The inevitable results
  - “Impedence mismatch” - in other words, BUGS!
  - Injection attacks and other security nightmares

JPA Fails (1)

- Configuration complexity
  - 108 annotation and enum classes
- A “query language” (actually more than one)
- No query performance transparency
  - Will `SELECT * FROM User ORDER BY lastName` be fast??
  - Multiple layers hide performance reality from the programmer
- Lack of data type congruence
  - `DATETIME != java.util.Date`
  - Floating point – there is no `NaN`
  - String truncation

JPA Fails (2)

- “Offline” Data – after the transaction has closed
- Imprecise control of what is kept in memory
  - JPA conflates “online cache” with “offline storage”
- After transaction closes, you can’t query
- No offline index information is kept
  - So queries would be slow even if you could do them

JPA Fails (3)

- Zero support for *schema management:*
- Schema verification
- Incremental schema evolution
  - `ALTER TABLE …` is a “stop the world” operation
- Structural vs. semantic schema updates
  - Structural: should be entirely automatic
  - Semantic: should be done at the language level
- Schema update type safety
  - Example: adding, renaming `Enum` constants

JPA Fails (4)

- Validation is not transactional
  - Foreign Key constraints apply on cache flush, not on commit
- Delete cascade can't handle cycles
- Data maintenance functions
  - At the database level instead of the language level
  - `SELECT name WHERE name.hashCode() = 17` ??
- Inconsistent handling of unique and `NULL`
- Cross-object validation is awkward

What is a Database?

- One or both of:
  - Hash table (unsorted keys)
    - Get, Put, Delete, Iterate
  - Balanced tree (sorted keys)
    - Find key via lower/upper bound
    - Iterate keys in sorted order
- Transactions
- A bunch of other convenience stuff
  - Data types (i.e., certain predefined encodings)
  - Automatically maintained indexes
  - SQL or other access language

JSimpleDB Approach

- The database is just a transactional, sorted key/value store
- JSimpleDB handles:
  - Encoding of data values (objects and fields)
  - Collection types (Set, List, Map)
  - Referential integrity
  - Indexes
  - Querying
  - Schema management
  - Validation

JSimpleDB Layering

- Key/Value Store Layer
  - `byte[]` keys and values
  - Provides transactions (and transaction semantics)
- Core API Layer
  - Data Encoding
  - Query views
  - Indexing
  - Schema management
- Java Layer
  - Maps Java model classes onto Core API
  - Wraps object ID’s with actual Java objects

Configuration

```
@JSimpleClass
public abstract class User implements HasAccount {

    @JField(indexed = true, unique = true)
    public abstract String getUsername();
    public abstract void setUsername(String username);

    public abstract String getEmail();
    public abstract void setEmail(String email);

    public abstract float getRanking();
    public abstract void setRanking(float ranking);

    public abstract Set<User> getFriends();
}

public interface HasAccount {
    Account getAccount();
    void setAccount(Account account);
}
```

Fields and Encoding

- Self-delimited encoding of all atomic values
  - For integers, optimized for small values (1 byte)
- Unique “storage ID” identifies types and fields
- Objects have unique 64 bit “object ID”
- Object ID’s assigned randomly, not sequentially
  - Avoids single point of contention

Key/Value Store Mapping

- ObjectID ➔ Object meta-data
  - E.g., schema version
- ObjectID + FieldID ➔ Field Value
  - No need to store default values
- ObjectID + FieldID + Index ➔ List Element
- ObjectID + FieldID + Set Element ➔ empty
- ObjectID + FieldID + Map Key ➔ Map Value

Key/Value Store Mapping - Indexes

- Create index by inverting field and value:
- FieldID + Field Value + ObjectID ➔ empty
- FieldID + List Item + ObjectID + Index ➔ empty
- FieldID + Set Element + ObjectID ➔ empty
- FieldID + Map Key + ObjectID ➔ empty
- FieldID + Map Value + ObjectID + Map Key ➔ empty

Reference Fields

- Always Indexed
  - Just like foreign keys in SQL
- Forward and Reverse delete cascades
  - Cycles are handled correctly
  - Possible to “ON DELETE REMOVE” from List
- Support for inverting reference paths
  - Inverting `children.element.friend` starting from X finds the parents of all for whom X is friend
  - This mechanism is used to implement `@OnChange`

Queries

- There are (only) three ways to start a query:
  - Find object by object ID ⇒ `T`
  - Find objects by Java type ⇒ `Set<T>`
  - Query an index ⇒ `Map<V,Set<T>>`
- What is an index on a field? A Map where:
  - Key = Field value
  - Value = Set of objects having that value in the field
- E.g. last name: `Map<String, Set<Person>>`
- Actually, NavigableSet and NavigableMap
  - Sorted, reversible, searchable

What about Joins?

- SQL joins are rooted in set theory
- Inner join = intersection
- Outer join = union
- Use regular Java collections
- Can iterate an intersection efficiently in O(n * m) time when the sets are sorted consistently, where n = # of sets and m = size of the *smallest* set. Provided by `NavigableSets.intersection()`.

Querying: The Price You Pay

- Programmer must now create a “query plan”
- In return: query performance transparency

```
// Get this user’s registration timestamp
public abstract Date getRegistrationDate();
public abstract void setRegistrationDate(Date date);

// Get users registered in the past day, sorted
// in reverse order of their registration time
public static Stream<User> getRegisteredToday() {

  // Get a view of the User.regDate index
  NavigableMap<Date, NavigableSet<User>> byRegDate
   = JTransaction.getCurrent()
    .queryIndex(User.class, "registrationDate", Date.class)
    .asMap();

  // Get all users registered in the past 24 hrs
  Date cutoff = new Date(
    System.currentTimeMillis() - 86400000);
  return byRegDate
    .tailMap(cutoff)
    .descendingMap()
    .stream()
    .flatMap(NavigableSet::stream);
}
```

Schema Management

- 100% schema verification
  - Active schema versions are recorded in the database
  - Incompatibilities are detected when transaction is opened
- Incremental schema evolution
  - Schema version is tracked on a per-object basis
  - “Rolling” application upgrades are possible
- Structural schema changes are fully automated
- Language-level semantic schema changes
  - `@OnVersionChange` methods handle any “fixups”
  - Previous version’s fields are made available
- **JSimpleDB always guarantees type safety**

Transactional Validation

- JSimpleDB maintains list of the object ID’s of any objects having pending validation
- Auto-detection of JSR 303 constraints
- You can manually add objects to the list as well
- `@OnValidate` for custom validation logic
- Actual validation is performed at commit time
  - Or any other time you want
  - Or never

Change Detection

- `@OnChange` methods invoked after any change in the specified object field(s)
- Fields may exist in objects reached through an arbitrary path of references
- Facilitates custom “indexes” (i.e., derived data)

@OnChange Custom Index Example

```
public abstract Set<Integer> getScores();

public double getAverageScore() {
    return (double)this.getSum().get() / this.getCount().get();
}

// Private fields
protected abstract Counter getSum();
protected abstract Counter getCount();

@OnChange("scores.element")
private void scoreAdded(SetFieldAdd<User, Integer> change) {
    this.getSum().increment(change.getElement());
    this.getCount().increment(1);
}

@OnChange("scores.element")
private void scoreRemoved(SetFieldRemove<User, Integer> change) {
    this.getSum().increment(-change.getElement());
    this.getCount().increment(-1);
}

@OnChange("scores.element")
private void scoresCleared(SetFieldClear<User> change) {
    this.getSum().set(0);
    this.getCount().set(0);
}
```

@OnChange Validation Example

```
// My salary
public abstract int getSalary();
public abstract void setSalary(int salary);

// My direct reports
public abstract Set<Employee> getReports();

// Invoked on my or any report's salary change
@OnChange({ "salary", "reports.element.salary" })
private void onReportSalaryChange() {
  this.revalidate(); // enqueue for validation
}

// Validate my salary vs. my direct reports.
// Invoked at end of transaction if my salary,
// or any of my reports' salaries, has changed
@OnValidate
private void checkSalaryInvariant() {
  int avgReportSalary = this.getReports()
    .stream()
    .mapToInt(Employee::getSalary)
    .average()
    .orElse(0);
  if (avgReportSalary > this.getSalary())
    throw new ValidationException("need a raise!");
}
```

Snapshot Transactions

- Offline data is stored in *snapshot transactions*
- A snapshot transaction is a regular transaction backed by an in-memory key/value store
- You explicitly specify what offline data to keep
- Indexes and querying function the same way
- Methods are provided to copy objects (in general, between any two transactions)
  - Data is copied efficiently at the key/value layer
  - Reference graph cycles are properly handled
- Snapshot transactions are handy for RPC as well
  - Same issues arise: encoding, schemas, indexes, querying

Command Line Interface

- Java 8 expression parser (method refs, etc.)
- Embeddable into application
- Pluggable CLI commands
- XML import/export
  - Key/value layer
  - Core API layer

Key/Value Implementations

- Third Party
  - Any SQL database (MySQL, CockroachDB, ...)
  - Berkeley DB Java Edition
  - FoundationDB
  - LevelDB, RocksDB
- JSimpleDB provided (serializable semantics)
  - Simple/flat file (uses read/read-write locking)
  - Read-optimized, memory mapped “array” k/v store
  - RaftKVDatabase distributed (Raft algorithm)

Links

- http://jsimpledb.org/ ...which redirects to...
- https://github.com/archiecobbs/jsimpledb
- Raft Consensus Algorithm: https://raft.github.io/