Reladomo: An Object Relational Mapping Framework

Agenda

- Chaining logic
- Object oriented, compiled time checked query language
- Transparent multi-schema support
- Object oriented batch operations
- Unit testable code
- Flexible object relationship inflation
- ...
- An introduction to Reladomo
- Why another persistence framework?
- Focus on "why" and "how" of various features
- User Driven Presentation: You decide the particular topics
- Future directions

Reladomo Features

- Metadata driven
- Code generation
- Object oriented query mechanism
- Junit integration
- Chaining
- Caching: Bi-level, transactionally guaranteed, keyless
- Flexible relationships
- Collections based operations (mass insert/update/delete; deep fetch)
- Multi-schema horizontally scaled databases
- Database vendor independence
- Temp Object (temp tables)
- Embedded Value Objects
- Natural handling of composite keys

Reladomo Features

- 2-tier and 3-tier (middle tier) operation
- Notification
- Primary Key Generation
- DDL Generator
- Database to XML Generator
- RUNS (Replication Update Notification System) integration
- Fast Sybase/UDB inserts
- Sybase bulk insert (pure Java)
- GS Integrator Transport
- Global Time support
- Documentation: javadoc, xsddoc, docbook
- Transaction support (local and 1.5 Phase XA via JOLT)
- Domain class diagram generation

Metadata Driven

Why?

- Declarative
- DRY Principle: adding an attribute should ideally be a single change
- Secondary uses: DDL generation, Visualization

How?

- XML
- Custom SQL-like relationship language

Metadata Driven

Example:

```
<MithraObject objectType="transactional"
>

  <PackageName>com.gs.fw.para.domain.desk.product</
PackageName>
        <ClassName>Product</ClassName>
        <DefaultTable>PRODUCT</DefaultTable>

  <SourceAttribute name="acmapCode" javaType="String"
/>

  <Attribute name="productId" javaType="int" columnName="P

  primaryKeyGeneratorStrategy="Max"
/>
```

Metadata Driven (Continued)

```
  <Attribute name="gsn" javaType="String" columnName="PROD_
/>

  <Attribute name="cusip" javaType="String" columnName="PRO
/>

  <Attribute name="issuerName" javaType="String" columnName
                   truncate="true"
/>

  <Attribute name="issuerNumber" javaType="int" columnName=
/>

  <Attribute name="description" javaType="String" columnNam
                   truncate="true"
/>
```

Metadata Driven (Continued)

```
  <Relationship name="synonyms" relatedObject="ProductSynon
to-many"

  reverseRelationshipName="product"
>ProductSynonym.productId = this.productId
        </Relationship>

  <Relationship name="history" relatedObject="ProductHistor
to-many"

  reverseRelationshipName="product"
>this.productId = ProductHistory.productId
        </Relationship>

  <Relationship name="currencySynonym" relatedObject="Produ
to-one"
>
            ProductSynonym.productId =
```

Metadata Driven (Continued)

```
            this.productId and ProductSynonym.type
 = "CID"
        </Relationship>

  <Relationship name="parentProduct" relatedObject="Product
to-one"
>
            ProductRelation.productChildId =
 this.productId
            and Product.productId =
 ProductRelation.productParentId and
 ProductRelation.relationshipType in (3200,
            3214, 9800, 3201, 3202, 3207,
            3208, 3209, 3210)
        </Relationship>
        <Index name="byGsn" unique="true"
>gsn</Index>
    </MithraObject>
```

Code Generation

Why?

- DRY: use the metadata to its fullest
- Quality: code written by domain experts
- Consistency: code is the same for all objects. Fixes/enhancements are propagated to all instances.
- Productivity: developers are freed to code the business logic instead of plumbing

How?

- JAXB XML parser: fast, easy to use
- Java based templates (similar to Eclipse JET): no need to learn another syntax. Supported by existing IDE’s (code completion, syntax highlighting, etc)
- JavaCC based relationship expression parser
- Extensible style code generation: generate abstract classes.

Object Oriented Query Language

- Compile time checked: if something changes, problems will be found earlier
- No strings: easy to reuse and abstract
- Overcomes some shortcomings of SQL: "Do not repeat yourself" (DRY) principle applied to relationships between objects
- Developers think in terms of objects and their relationships, not tables and joins.
- In-line SQL is difficult to write, harder to reader and nearly impossible to maintain
- In-line SQL is difficult to abstract and reuse
- String based solutions (e.g. HQL, OQL, EQL, etc) do not solve these issues
- Reladomo uses an object oriented query language that fits comfortably within the programming environment:

In-line SQL Example

```
public void selectTransactions()
throws TransactionQueryException, SQLException,
 CriteriaException
{
    this.createUpdateStatementWrapper();
    try
    {
        StringBuffer sb = new StringBuffer();
        sb.append(" select BTV.*, NPV.NPV, F.RATE,
 NPV.RAW_UNREAL, NPV.DISC_UNREAL, NPV.ADJ_NPV ");
        sb.append(" into #tran_union ");
        sb.append(" from BASIC_TRANSACT_VIEW BTV,
 #accounts A, FX_FORWARD_NPV NPV,
        SECDB_FOREX_RATE F, TCURRENCY C ");
        sb.append(" where BTV.ACCT_ID = A.ACCT ");
        sb.append(" and BTV.TRAN_ID = NPV.TRAN_ID
 ");
```

In-line SQL Example (Continued)

```
       sb.append(" and BTV.TRUE_STATUS = 'ACTIVE'
");
       sb.append(" and BTV.OUT_Z >= ? ");

 this.addTimestampParameter(this.getBasicDateProvider().fe
       sb.append(" and BTV.IN_Z < ? ");

 this.addTimestampParameter(this.getBasicDateProvider().fe
       sb.append(" and BTV.TRAN_SETTLE_D > ? ");

 this.addTimestampParameter(this.getBasicDateProvider().ge
       sb.append(" and NPV.FROM_Z < ? ");

 this.addTimestampParameter(this.getBasicDateProvider().ge
```

In-line SQL Example Continued

```
sb.append(" and NPV.THRU_Z >= ? ");
this.addTimestampParameter(this.getBasicDateProvider().getA
sb.append(" and NPV.IN_Z < ? ");
this.addTimestampParameter(this.getBasicDateProvider().getE
sb.append(" and NPV.OUT_Z >= ? ");
this.addTimestampParameter(this.getBasicDateProvider().getE
sb.append( "and F.CURRENCY = 'USD' ");
sb.append(" and BTV.TRAN_SETTLE_D = F.VALUE_DATE");
sb.append(" and F.FROM_Z < ? ");
this.addTimestampParameter(this.getBasicDateProvider().getA
sb.append(" and F.THRU_Z >= ? ");
this.addTimestampParameter(this.getBasicDateProvider().getA
sb.append(" and F.IN_Z < ? ");
this.addTimestampParameter(this.getBasicDateProvider().getE
sb.append(" and F.OUT_Z >= ? ");
this.addTimestampParameter(this.getBasicDateProvider().getE
sb.append(" and BTV.PROD_SEC_ID_I =
 C.PROD_SEC_ID_I");
this.getStatementWrapper().setStatementString(sb.toString()
```

In-line SQL Example Continued (Continued)

```
this.executeUpdateStatement();
}
catch (DataStoreException e)
{
this.getLogger().error(e);
}
}
```

Object Oriented Query Example

```
public List buildOperation
 (PnlObjectOperationProvider pnlObjectOpProvider,
 ProductOperationProvider
productOpProvider, ParaDate milestoneBusinessDate,
 ActivityReviewManager activityReviewManager)
{
    this.activityReviewManager =
 activityReviewManager;
    ParaTransactionList basicTranList = new
 ParaTransactionList();
    List tranList
 = this.buildBusinessDateBasicTransactionOperation(pnlObjec
 productOpProvider,
    milestoneBusinessDate);
    for(int i = 0; i< tranList.size(); i++)
    {
        ParaTransactionList itemList =
 (ParaTransactionList)tranList.get(i);
```

Object Oriented Query Example (Continued)

```
        Timestamp busDate = new
 Timestamp(milestoneBusinessDate.getTime());
        businessDate = busDate;
        Operation op =
 itemList.getOperation().and(ParaTransactionFinder.status()

 .and(ParaTransactionFinder.settleDate().greaterThan(busDat

        op =
 op.and(getStringOperation(getActivityReviewManager().getCo
        basicTranList.add(new
 ParaTransactionList(op));

 basicTranList.deepFetch(ParaTransactionFinder.underlierTra

 basicTranList.deepFetch(ParaTransactionFinder.customerTran
    }
    return basicTranList;
}
```

Object Oriented Query Language

How?

- Atomic (equals, in, greaterThan, lessThan, etc)
- Mapped (traversing a relationship, aka join)
- Boolean (and, or)
- Miscellaneous (all, absolute value, etc)

- Non-trivial: Large part of the Reladomo code base (> 20%)
- Various types of operations
- Before evaluation of a complex operation, it’s simplified.
- Operation is evaluated against the cache (if applicable) and then the server
- SQL generation can be a bit tricky (especially for dated objects)

Testable Code

Why?

- Testable code has become an indispensable part of our development methodology
- Persistent objects are traditionally difficult to unit test because they’re tied to a database
- The core Reladomo code was written using test driven development

How?

- The crux of the code is processing of data.
- Reladomo-enabled testing covers > 80% of the code.
- Result: shortened development time, highly reliable code with very few bugs encountered in production.
- Create a test resource: text file for initial data + in memory SQL database (H2)
- Reladomo provides a simple testing framework that fits right into Junit.

Testable Code (Continued)

- All operations are supported: query, insert, update, delete, chaining, etc.
- Examples: Large production application

Flexible Relationships

Why?

- Relationships between objects can take interesting forms in real life.
- Can dramatically reduce IO to the database. Can also be used for interesting searches.
- Two common examples: a parametrized relationship, or a relationship with extra conditions.

How?

- This feature works because of Reladomo’s dynamic relationship resolution. Examples: Relationships from Product

```
<Relationship name="parentProduct" relatedObject="Product
to-one"
>
    ProductRelation.productChildId =
 this.productId and Product.productId =
    ProductRelation.productParentId and
```

Flexible Relationships (Continued)

```
    ProductRelation.relationshipType in (3200,
 3214, 9800, 3201, 3202, 3207, 3208, 3209, 3210,
 3211)
</Relationship>
<Relationship name="synonymItem" relatedObject="ProductSy
to-one"
              parameters="String sym"
>
    ProductSynonym.productId = this.productId and
 ProductSynonym.type = {sym}
</Relationship>
```

Chaining

Chaining is an umbrella term that describes a way of storing time series data, audit data or both in a relational database. The different versions (audit only, time series only and bitemporal) are described below.

- 1.Audit Only
- 2.Business Time Series Only
- 3.Both Audit and Business Time Series: Bitemporal

Chaining

Why?

- Chained objects are queried and persisted differently
- Chained objects don’t have the same operations (insert, update, delete) as regular objects
- Chained objects support more complicated operations: insert, insert until, update, update until, increment, increment until, terminate.
- Chaining is complicated
- The algorithm is only maintainable if it’s managed from one single piece of code
- Chaining affects the core of object-relational mapping. It is very difficult to implement chaining as an add-on to an existing OR framework.

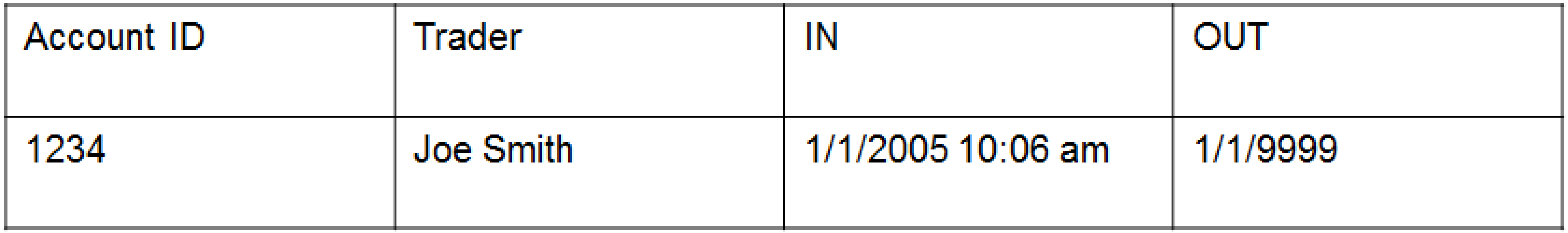
Chaining

How?

- Not a large piece of code (6%), but complicated: 30% of test code is just for chaining
- Information held in a single object is usually not enough to calculate new state
- Object delegates complex operations to the TemporalDirector
- TemporalDirector uses TemporalContainer to calculate new state
- TemporalContainer keeps data for a range of dates. Can fetch more from the database, on demand.
- TemporalContainers are held in the transactional cache and discarded at end of transaction

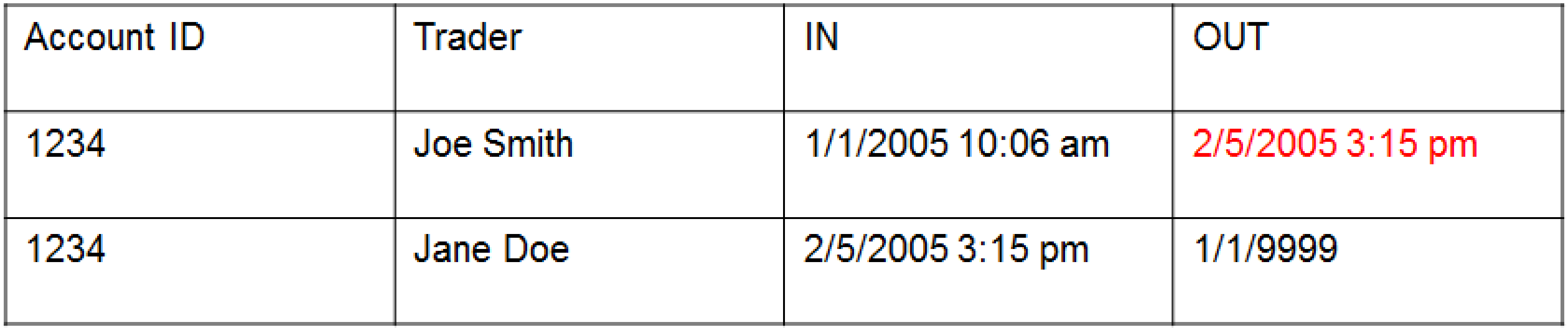
Audit Only Chaining

Here is an example of this type of audit trail for an account object. The account was created on 1/1/2005:



Account ID	Trader	IN	OUT
1234	Joe Smith	1/1/2005 10:06 am	1/1/9999

On 2/5/2005, the trader changes to Jane Doe:

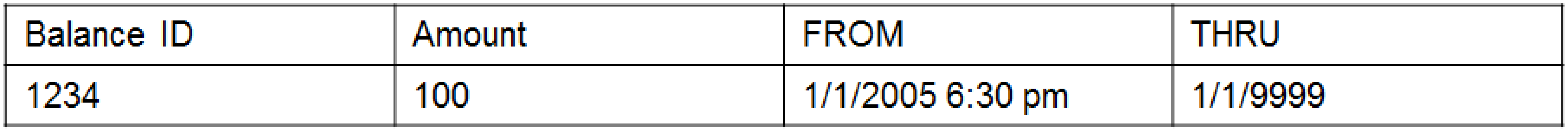


Account ID	Trader	IN	OUT
1234	Joe Smith	1/1/2005 10:06 am	2/5/2005 3:15 pm
1234	Jane Doe	2/5/2005 3:15 pm	1/1/9999

Audit Only Chaining

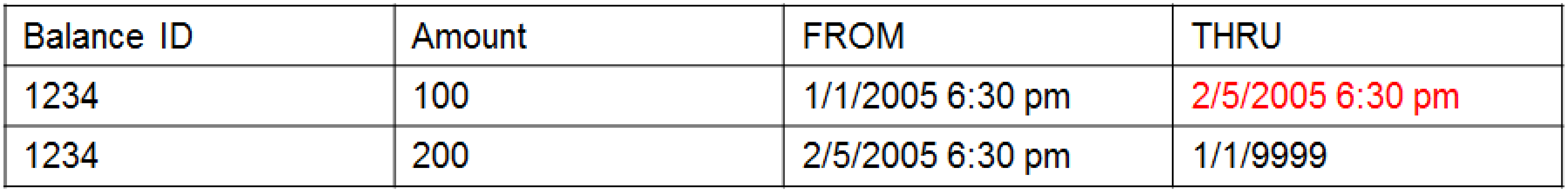
- The IN and OUT columns represent real time. They have nothing to do with the business calendar.
- The interesting row (meaning, the row we think has the correct information) always has OUT = Infinity
- There is no way to alter the history. The only allowed update operation to a row is to change its OUT value from infinity to current time.

On 1/1/2005, we buy 100 shares of a product. We always do our accounting at 6:30 pm (even if it takes several hours, our business calendar is set to 6:30 pm):



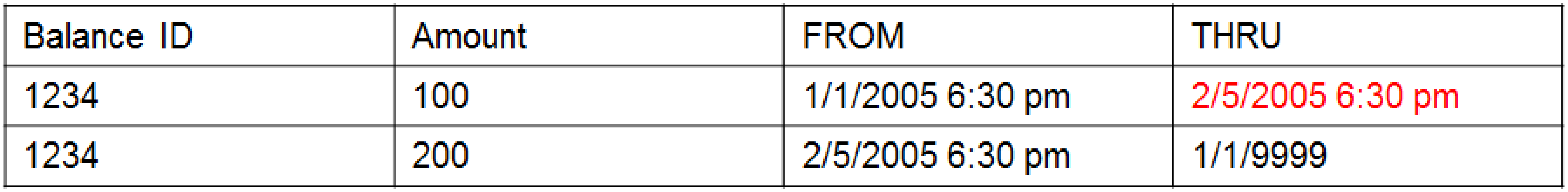
Balance ID	Amount	FROM	THRU
1234	100	1/1/2005 6:30 pm	1/1/9999

On 2/5/2005, we buy another 100 shares:



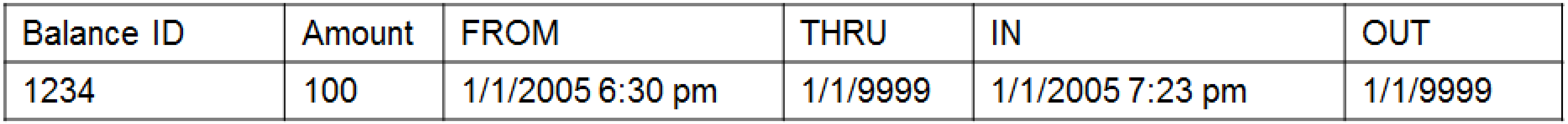
Balance ID	Amount	FROM	THRU
1234	100	1/1/2005 6:30 pm	2/5/2005 6:30 pm
1234	200	2/5/2005 6:30 pm	1/1/9999

So far, this looks very much like the first example. To clarify the difference, we can do an "as of trade". On 2/10/2005, we find out that we missed a trade for 50 shares that happened on 1/15/2005:



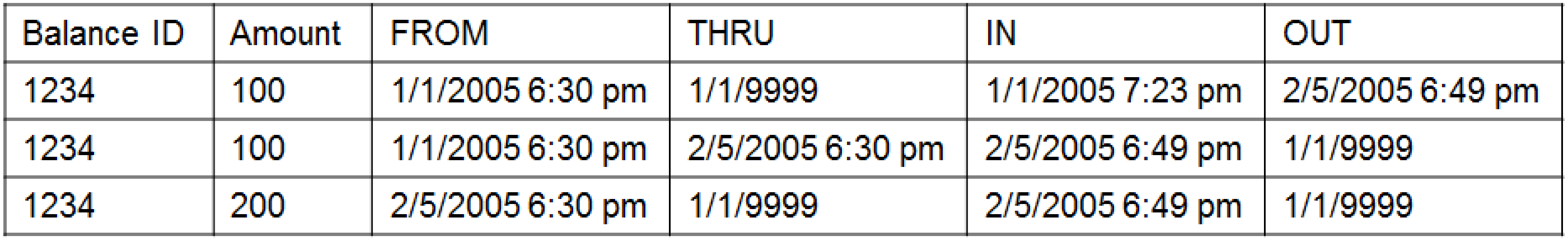
Balance ID	Amount	FROM	THRU
1234	100	1/1/2005 6:30 pm	2/5/2005 6:30 pm
1234	200	2/5/2005 6:30 pm	1/1/9999

Let’s consider the same example



Balance ID	Amount	FROM	THRU	IN	OUT
1234	100	1/1/2005 6:30 pm	1/1/9999	1/1/2005 7:23 pm	1/1/9999

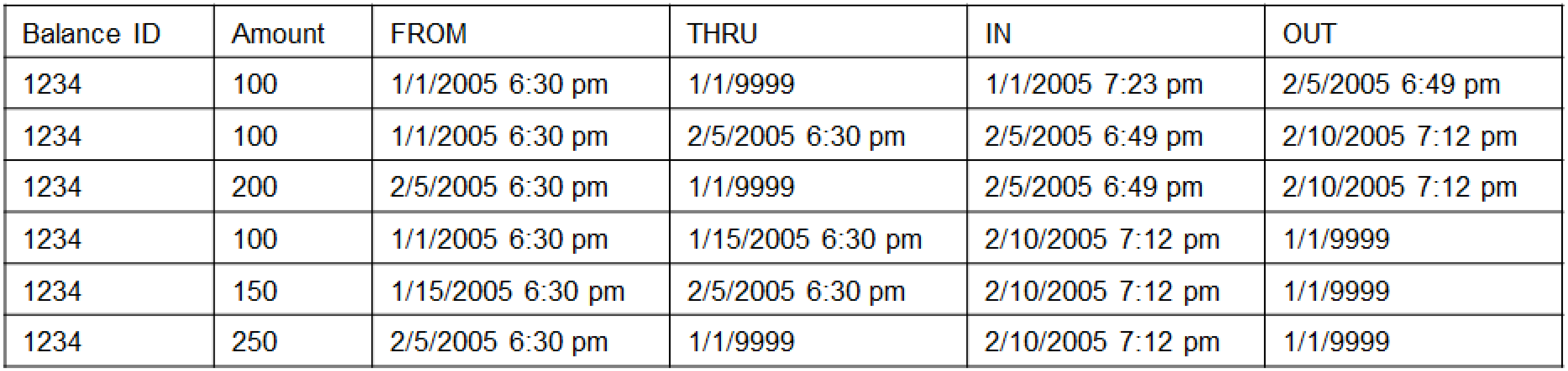
We now add 100 on 2/5/2005:



Balance ID	Amount	FROM	THRU	IN	OUT
1234	100	1/1/2005 6:30 pm	1/1/9999	1/1/2005 7:23 pm	2/5/2005 6:49 pm
1234	100	1/1/2005 6:30 pm	2/5/2005 6:30 pm	2/5/2005 6:49 pm	1/1/9999
1234	200	2/5/2005 6:30 pm	1/1/9999	2/5/2005 6:49 pm	1/1/9999

On 2/10/2005, we find a trade that was done on 1/15/2005 for 50 shares:

(Continued)



Balance ID	Amount	FROM	THRU	IN	OUT
1234	100	1/1/2005 6:30 pm	1/1/9999	1/1/2005 7:23 pm	2/5/2005 6:49 pm
1234	100	1/1/2005 6:30 pm	2/5/2005 6:30 pm	2/5/2005 6:49 pm	2/10/2005 7:12 pm
1234	200	2/5/2005 6:30 pm	1/1/9999	2/5/2005 6:49 pm	2/10/2005 7:12 pm
1234	100	1/1/2005 6:30 pm	1/15/2005 6:30 pm	2/10/2005 7:12 pm	1/1/9999
1234	150	1/15/2005 6:30 pm	2/5/2005 6:30 pm	2/10/2005 7:12 pm	1/1/9999
1234	250	2/5/2005 6:30 pm	1/1/9999	2/10/2005 7:12 pm	1/1/9999

Collections Based Operations

- Prepared statement batching: reuse the same statement multiple times. X 2 performance improvement
- Use of SQL statements that update more than one row at a time. X 50 performance improvement
- Two types of batching:
- Collections are a core of the Reladomo API.
- Example mass delete:

```
Operation op =
 SwapPriceFinder.sourceId().eq( id ); op =
 op.and(

 SwapPriceFinder.businessDate().eq( busDate ) );

op =
 op.and( SwapPriceFinder.feedNumber().eq( feedNumber ) );

SwapPriceList priceList = new SwapPriceList(op);
```

Collections Based Operations (Continued)

```
priceList.deleteAll();
```

- 65,583 rows took 562 seconds without deleteAll implementation. With the implementation it took 12 seconds.
- Deep Fetching: a better approach to relationship resolution
- Collections based operations make Reladomo suitable for most types of large retrievals (report style), OLTP, and batch style processing.

Collections Based Operations

Why?

- Reduce object relational impedance mismatch
- Reduced chattiness
- Performance

How?

- Investigating pure Java alternative to file generation
- List object used as gateway to collective operations
- Special SQL generation for mass/bulk operations
- Deep fetch uses joins: solves 1+N problem
- BCP support for Sybase: 5x faster than plain insert

Transparent Multi-schema Support

- For scalability, we’ve partitioned ledger data into a large number of databases (about 150). The schema is identical in these database, but the data is different.
- The class of objects can therefore be retrieved from multiple sources
- Traditional ORMs have difficulty keeping objects tied to the original source. This is particularly a problem with caching.
- We even have transactions that read from one database and write to another. That is, the access patterns are not necessarily one-database-at-a-time.
- Support for this is built into the core of Reladomo.

Transparent Multi-schema Support

Why?

- Transaction 123 in Database A can be 100 shares of IBM
- Transaction 123 in Database B can be 300 shares of BMW
- When both objects are loaded, they must not be confused.
- How an object is identified must include where the object came from:
- Enables horizontally scalable solutions

How?

- Metadata includes special attribute (SourceAttribute)
- All operations (find, insert, update, delete) use this attribute to obtain the proper connection.

Caching

Why?

- Uniquing: an object with a given primary key must correspond to exactly one memory location
- Performance
- Reduced IO and latency

How?

- Can be configured as none, partial (dynamic) or full on a per class basis.
- Can be bypassed on a per query basis.
- Cache is a searchable set of indices. An index is a keyless set.
- Queries are cached in the query cache. Also facilitates deep fetched relationships.
- Transaction disregards pre-transaction cached results.
- Partial cache can only answer queries based on unique identifiers.

Three Tier Operation

Why?

- User ID must not be able to access database directly (especially write)
- Batch/App ID must not be used from unauthorized IP’s (see PACT AppFilter)
- For a large, semi-mobile user community, maintaining IP lists is undesirable and opens iSQL hole
- Security (fat client applications):
- Connection sharing: database connections can be expensive. Many users can share same connection.

How?

- Third tier acts like a relational source. Supports relational-like operations: find, insert, update, delete.
- No object graphs. Not a complex object source. Serializaton based on metadata. Wire format looks like a result set.
- Lightweight: can be configured as pass-through with no caching.

Three Tier Operation (Continued)

- Remoting API must be implemented by application.

Notification

Why?

- Allow multiple VM’s to independently update data.
- Polling considered harmful ("Are we there yet?" syndrome)

How?

- At the end of a transaction, message is constructed. Message contains the primary keys for objects that were inserted/updated/ deleted. Message is sent to a topic that encodes the database identity.
- Listeners only register interest in databases they have accessed. Upon receipt of message, any objects (if any) are marked as dirty.
- Asynchronous message processing to avoid messaging and IO bottlenecks in application’s main flow.
- Messaging API can be implemented by application. RV implementation provided.
- Notification is entirely independent of three tier operation. Most important production uses are in two tier scenarios. Notification is off by default.

Notification

Examples:

- Posting Engine creates an account. Adjustment server processes a request for the trial or income function containing the new account some time later.
- Age Inventory Firm to Firm processor on Desk A updates age transfer status. Age Inventory Firm to Firm processor on corresponding desk will see new status.
- Posting Engine updates feed status. Notification is sent for the status object. Next time a controller queries for status, they will not get stale results.

RUNS Integration

Why?

- Replication from remote sources can cause staleness.
- For low volume update data (e.g. account data) hitting database all the time is wasteful.
- Object metadata can be used the same way with RUNS tables as regular tables.
- Staleness typically exasperated because objects are configured as read only.

How?

- Application configuration flags objects that are replicated.
- Background thread reads RUNS queue tables periodically.
- Send notification based on primary key found in RUNS child tables
- Clear RUNS tables.
- Fully optional. Can be setup as a lightweight, independent process.

DDL Generator

Why?

- Metadata contains all necessary data. DRY: get the DDL from the metadata.
- Productivity: DDL files are hard to write and maintain.
- Junior developers have problems writing DDL files, especially index creation.

How?

- Based on the metadata and target database type, emit DDL.
- Hardest part is generating decent indices. Primary key index is easy. Foreign key indices are based on defined relationships.

Generate metadata from existing schema

Why?

- Large legacy systems can be converted quickly and painlessly.

How?

- Create object definition from table definition.
- Choose object primary key based on unique index.

Long term plan

- Feedback is the most valuable thing. What’re we doing right or wrong?
- What features would make your code better?
- If you find a bug, a test case would be exceptionally helpful.
- If you’re feeling adventurous, contribute code!
- The direction of Reladomo is set by its users.
- Help us make Reladomo a better product: