Reladomo Advanced Use Cases

- Small Features
- Dated Object Features
- forEachWithCursor
- Modifiable (Mutable) Primary Keys
- Detached Objects
- Aggregation
- Temp Object
- Tuples
- Setting up a Notification Server
- Update Listener
- Multi-Threaded Matcher Loader

Small Features

- attribute.in(list, attr)

```
ProductFinder.findMany(
    ProductFinder.productId()
        .in(paraTranList,
 ParaTransactionFinder.productId())
```

- attribute.filterEq

```
PositionCarryFinder.income().filterEq(
    PositionCarryFinder.expense());
```

SQL:

```
SELECT *
FROM TPOSFUNDING_CRY t0
WHERE t0.INCOME = t0.EXPENSE
```

- deleteInBatches

```
FooList list = FooFinder.findMany(op);
```

Small Features (Continued)

```
list.deleteAllInBatches(10000);
can also handle arbitrary list of objects:
list = new FooList();
list.add(...);
list.deleteAllInBatches(50000);
```

Advanced Dated Object Features

- Use these only for archival or special needs: In-place update, purge, insertForRecovery
- Must not be used in normal course of work
- In-place update for dated objects

  Must mark the attribute in the xml:

```
<Attribute name="description" javaType="String"
    columnName="DESCRIPTION" maxLength="50"
    truncate="true" inPlaceUpdate="true"
>
```

- Must call the setInPlace method:

```
public void
 setDescriptionUsingInPlaceUpdate(String
 newValue)
```

- Purge

Advanced Dated Object Features (Continued)

- Can be used to physically delete a dated object
- Normally, deletes the entire history (past and future)
- When coupled with equalsEdgePoint, can delete specific range of history, or just one row.

```
object.purge(); // deletes all of the object's
 history
list.purgeAll(); // deletes everything specified
 by the list's operation
```

Advanced Dated Object Features

- insertForRecovery

  can be used to insert an object with arbitrary in/out

```
PositionQuantity pos = new
 PositionQuantity(infinity);
pos.setProcessingDateFrom(...);
pos.setProcessingDateTo(...);
pos.set...

pos.insertForRecovery();
```

forEachWithCursor

- Normally:

```
ProductList list = ProductFinder.findMany(op);
for(int i = 0; i < list.size(); i++)
{
    Product prod = list.get(i);
    ...
}
```

- Instead:

```
list.forEachWithCursor(new TObjectProcedure()
{
    public boolean execute(Object o)
    {
        Product prod = (Product) o;
        // do something with prod,
        // but don't call any relationship
 methods on it
```

forEachWithCursor (Continued)

```
        return true; // return false would end
 the loop.
    }
});
```

forEachWithCursor

- Normally, we read all products for that list into memory on the first call to size() (or any other methods, like get());
- forEachWithCursor allows looping through the results one at a time before all the results are loaded
- Advantages:
  - Can process data before query is finished: potentially faster processing
  - Less memory requirement: Reladomo won't hold onto the object, so very large result sets can be processed
- Disadvantages:
  - Does not support deep fetch
  - Does not cache the results, which can be bad if the query is repeated

Detached Objects

- Detached Objects
  - Delayed edit functionality; useful for GUI, where the user is modifying objects but can choose to Save or Cancel
  - Don't use this for a transactional scenario, where object is read and modified in the transaction and there is no "cancel"
  - Cannot modify the primary key of a detached object

```
Product prod = ProductFinder.findOne(
    ProductFinder.productId().eq(12));
Product detachedProd = prod.getDetachedCopy();
// while the user is editing:
detachedProd.setDescription("something new");
// does not write to the database
detachedProd.set...

// after the user presses the save button
detachedProd.copyDetachedValuesToOriginalOrInsertIfNew();
```

Detached Objects

- Can handle full object graph of dependent objects:
  - Must mark relationship in object as "relatedIsDependent="true""

TraderPack.xml:

```
<Relationship relatedObject="Section"
    relatedIsDependent="true"
    cardinality="one-to-many"
    name="sections"
>
    this.traderPackId = Section.traderPackId
</Relationship>
```

- Code:

```
TraderPack detachedPack = pack.getDetachedCopy();
SectionList sections =
 detachedPack.getSections();
// returns a detached list of sections
sections.get(0).setSectionName("new name");
```

Detached Objects (Continued)

```
// does not write to the database
sections.remove(2);
// remove the 3rd element of the list
Section newSection = new Section();
newSection.set...
sections.add(newSection);

// user presses the save button:
detachedPack.copyDetachedValuesToOriginalOrInsertIfNew();
// saves the pack and its sections.
// removed sections are deleted. New sections are
 inserted
```

Detached Objects

Useful methods:

```
isModifiedSinceDetachment();
```

```
isModifiedSinceDetachmentByDependentRelationships();
```

- Compare all non-primary key values
- Returns true if dependent relationship is modified
- ```
  isModifiedSinceDetachment(Extractor extractor);
  ```
- ```
  detachedProduct.isModifiedSinceDetachment(
      ProductFinder.description());
  ```
- Same as above, but for relationship

  ```
  isModifiedSinceDetachment(
      RelatedFinder relationshipFinder);
  ```
- Resets the detached object to the original values

  ```
  resetFromOriginalPersistentObject();
  ```

Aggregation

- Equivalent to SQL group by
- Allows aggregate functions: sum, avg, min, max, count
- Example:

```
AggregateList aggList = new AggregateList(op);
//the op determines the where clause
aggList.addGroupBy(
    "acct", PositionCarryFinder.accountId());
// can call addGroupBy multiple times
aggList.addAggregateAttribute(
    "lastProcTime",

 PositionCarryFinder.processingDateFrom().max());
aggList.addAggregateAttribute(
    "count",
    PositionCarryFinder.accountId().count());
aggList.addAggregateAttribute(
    "income",
    PositionCarryFinder.income().sum());
```

Aggregation (Continued)

```
for(AggregateData data: aggList)
{
    String accountId =
 data.getAttributeAsString("acct");
    int count = data.getAttributeAsInt("count");
    double income =
 data.getAttributeAsDouble("income");
}
```

- Can do a bit of math (plus, minus, times, divide):

```
aggList.addAggregateAttribute(
    "incExp",
    PositionCarryFinder.expense()

 .plus(PositionCarryFinder.income()).sum());
```

Temp Objects

- Usually used for driver of some kind
- Must pre-define xml:

```
<MithraTempObject>

  <PackageName>com.gs.fw.para.domain.desk.transaction
    </PackageName>
    <ClassName>DividendPositionDriver</ClassName>


  <SourceAttribute name="acmapCode" javaType="String"
/>
    <Attribute name="accountId" javaType="String"
        primaryKey="true" maxLength="20"
/>
    <Attribute name="productId" javaType="int"
        primaryKey="true"
/>
```

Temp Objects (Continued)

```
</MithraTempObject>
```

- Add xml to the class list at the end:

```
<MithraTempObjectResource name
 = "DividendPositionDriver"

/>
```

- Add the temp object to the runtime configuration

Temp Objects

In code, create a temporary context, insert some values, then join to the destination table

```
TemporaryContext positionDriverContext =
    DividendPositionDriverFinder.
createTemporaryContext
(getDeskAcmapCode());
try
{
    DividendPositionDriverList tempList =
        new DividendPositionDriverList();
    tempList.add(...);
    tempList.insertAll();
    Operation op =
 ParaTransactionFinder.acmap().eq("VOL");
    op = op.and(

 ParaTransactionFinder.type().beginsWith("SWP"));
    op = op.and(
```

Temp Objects (Continued)

```
        DividendPositionDriverFinder.
existsWithJoin
(
            PTF.acmap(), PTF.accountId(),
 PTF.productId());
    // PTF == ParaTransactionFinder
    ParaTransactionList tranList =
        ParaTransactionFinder.findMany(op);
    // use the list
}
finally
{
    positionDriverContext.
destroy
();
}
```

Modifiable (Mutable) Primary Keys

- Only sensible for composite keys
- Must mark the attributes in XML:

```
<Attribute name="currency" javaType="String"

  columnName="PROD_CURRENCY_C" primaryKey="true"
    trim="true" maxLength="3"
/>
<Attribute name="source" javaType="int"
    columnName="SOURCE_I" primaryKey="true"

mutablePrimaryKey
="true" nullable="true"/>
<Attribute name="date" javaType="Timestamp"
    columnName="THRU_Z" primaryKey="true"

mutablePrimaryKey
="true"/>
```

Modifiable (Mutable) Primary Keys (Continued)

- setSource and setDate methods are now allowed to be called on a persisted object

  setCurrency is not allowed
- SQL looks like:

```
update FXRATE

set SOURCE_I = 12

where PROD_CURRENCY_C = 'USD'
and
SOURCE_I = 10

and THRU_Z = '2008-03-10 00:00:00'
```

Tuples

- Occasionally, it’s very useful to be able to do large in-clauses with combination of attributes (aka a "tuple")
- The API is simple, just two methods: ***tupleWith()*** and ***in()***
- First, we have to create a tuple using the "tupleWith" method on a normal Attribute

  For Example:

```
TupleAttribute tupleAttribute =
    PositionFinder.accountId().tupleWith(
        PositionFinder.productId());
```

- We can keep on adding more attributes with tupleWith
- There are various forms of "in" that can then be used with the tuple attribute to create an operation.
- Create a MithraArrayTupleTupleSet and call in(set)
- Use the (list, Extractor[]) form, just like a regular attribute in(list, Extractor)
- Use the (aggregateList, String… aggregateAttributeName) for some advanced use of aggregation combined with a normal query

Setting up a TCP Notification Server

- Reladomo supports multiple notification mechanisms
- The TCP notification server is the easiest to setup and works well for a small number of servers and clients (less than 1000)
- There are two steps required for this:
  - Setup the notification server:

```
java –classpath <all_the_required_jars>
    –Dport=<some_port_number>
    com.gs.fw.common.mithra
        .notification.server.NotificationServer
```

  - In the processes that read or write Reladomo objects, configure them for notification:

```
MithraManagerProvider.getMithraManager()
    .setNotificationEventManager(
        new MithraNotificationEventManagerImpl(
            new
 TcpMessagingAdapterFactory(host, port)));
```

Setting up a TCP Notification Server (Continued)

- The host and port should point to where the NotificationServer is running

Update Listener

- Add the update listener in the object XML:

```
<MithraObject objectType="transactional"
>

<PackageName>com.gs.fw.common.mithra.test.domain</
PackageName>
<ClassName>Division</ClassName>


        <UpdateListener>
         com.gs.fw.common.mithra.test.domain.DivisionUpda
        </UpdateListener>

<DefaultTable>DIVISION</DefaultTable>
```

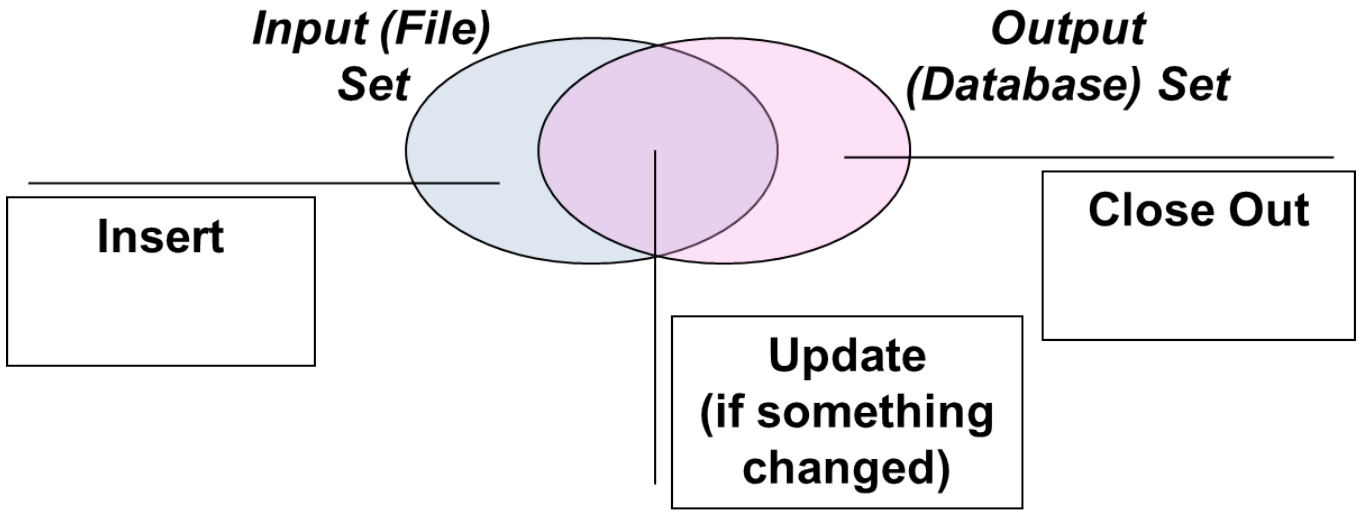
- Implementation must have an empty constructor and implement MithraUpdateListener

Update Listener (Continued)

```
public void handleUpdate(T updatedObject,
 UpdateInfo updateInfo);
public void handleUpdateAfterCopy(T
 updatedObject);
```

- Only called on persistent objects
- Typical use case: set the "changed by" field

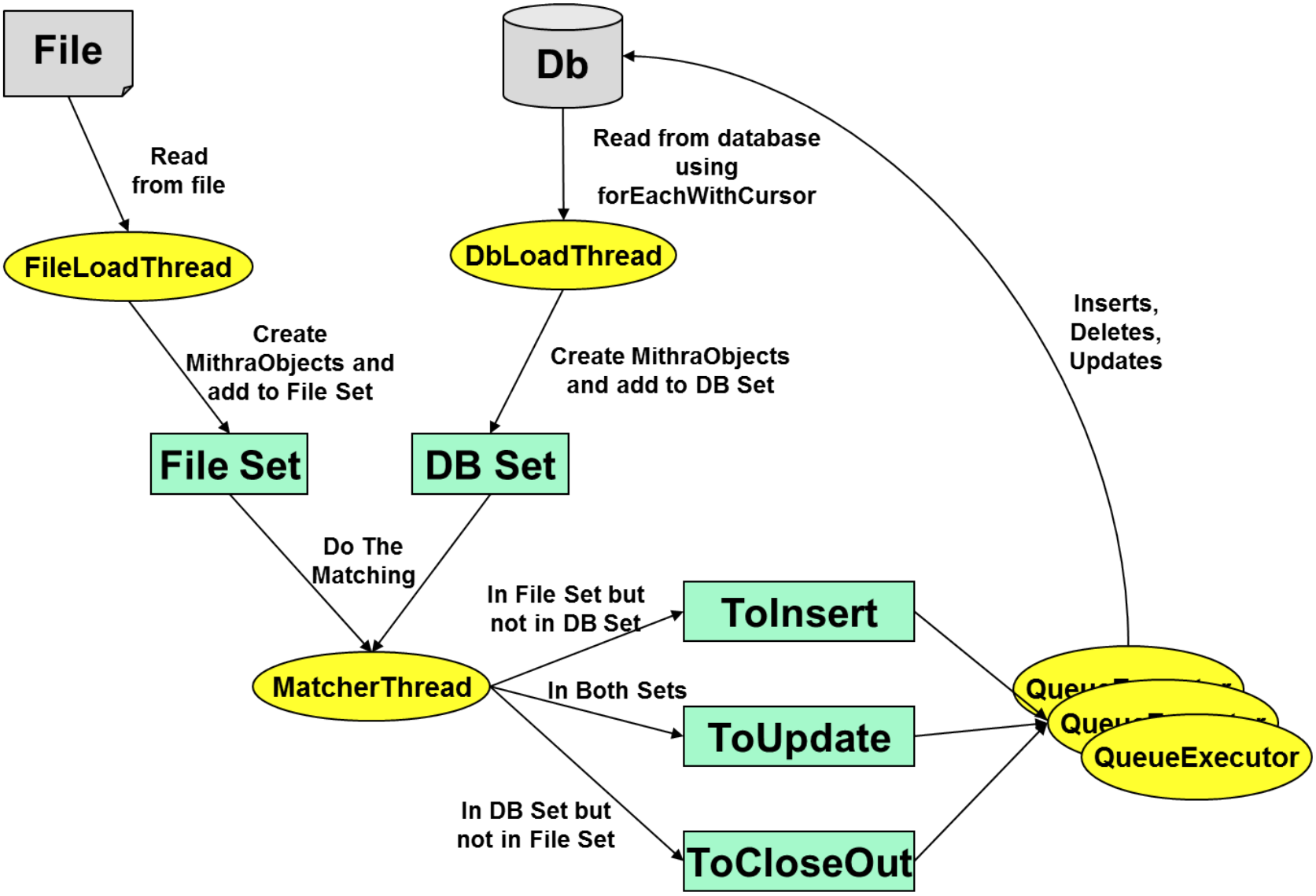
Multi-Threaded Matcher Loader



Schematic Representation

- The application finds the intersection of the two Sets
- Whatever is in the intersection, will be updated (but only if something changed)
- Whatever in in Input Set but not in Output Set will be inserted
- Whatever is in Output Set but not in Input Set will be closed out (deleted or terminated)

Multi-Threaded Matcher Loader



Multi-Threaded Matcher Loader (Continued)

Multi-Threaded Matcher Loader Architecture

Multi-Threaded Matcher Loader

- Highly customizable behavior
  - Doesn’t have to be File-to-Database; can easily be Database-to-Database or Memory-to-Database, etc.
  - Subclassing MatcherThread or SingleQueueExecutor allows fine tuning for different requirements
  - For example, can decide not to delete the left overs.
- By design, it is re-runnable
- Usable under the following conditions:
  - No transactional guarantee or ordering required
  - Writing to one table only. Writing to multiple tables is possible but without transactional guarantee.
- Classes to know
  - SingleQueueExecutor – the piece of code that does the actual writing
  - MatcherThread – the piece of code that matches the two sets
  - DbLoadThread – the piece of code that reads the database set, using forEachWithCursor

Multi-Threaded Matcher Loader (Continued)

- InputThread – the piece of code that generates the input, e.g. from a file.

But Wait... There Is More

- Time-Zone Conversion
- Nested Reladomo Test Resource
- Optimistic Locking
- Partial Cache Tuning
- Class Diagram Generation
- DDL Generation