

# Advanced ColdFusion-ORM

#adobemax348

Rupesh Kumar



[www.rupeshk.org/blog](http://www.rupeshk.org/blog)

Senior Computer Scientist  
Adobe Systems Inc.

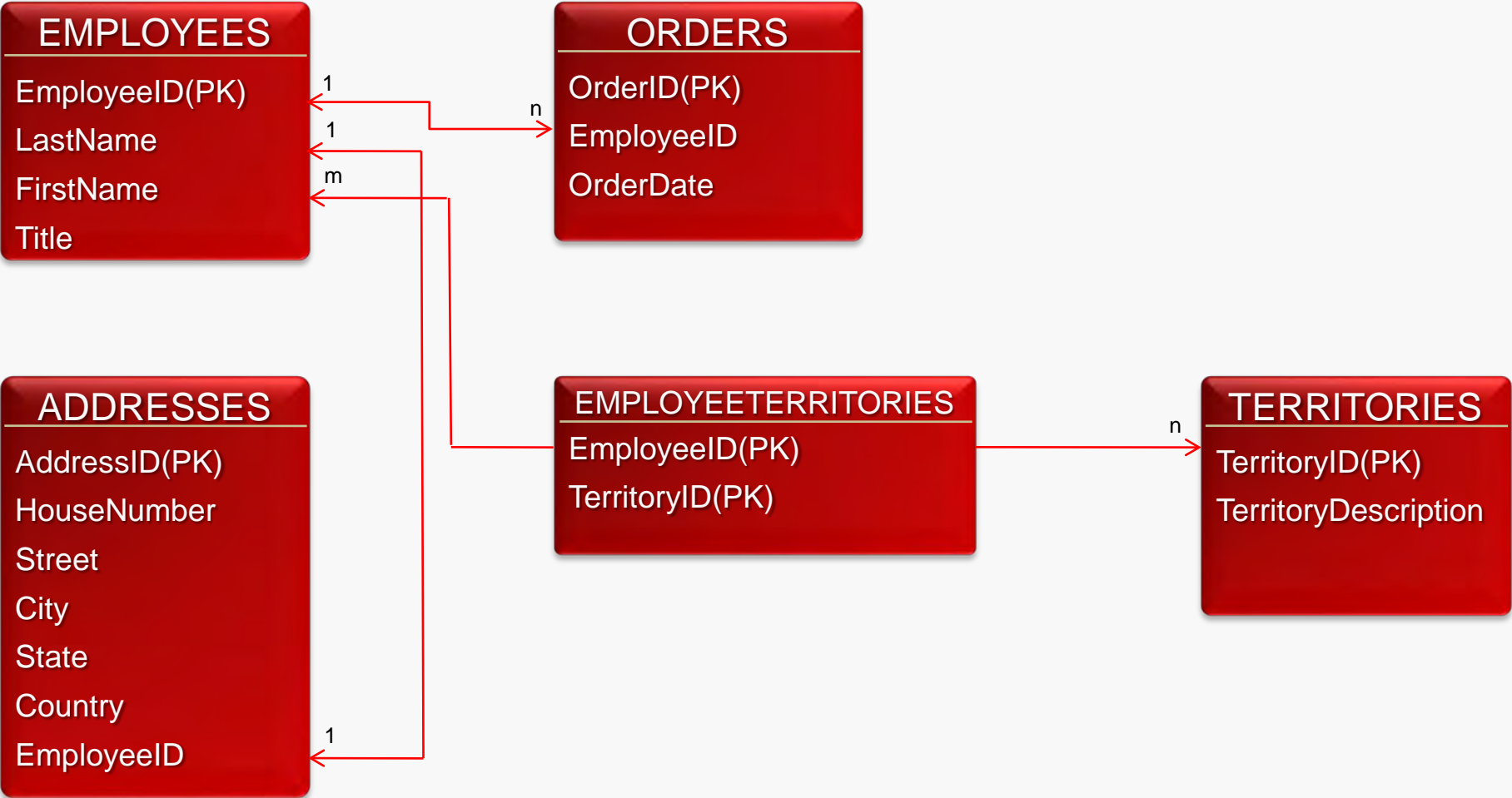


rukumar



- ORM Basics
- Demo
- Advanced Mapping
- What happens internally
- Transaction & concurrency control
- Caching & optimization
- Event Handling
- Auto-generation of tables
- Q & A

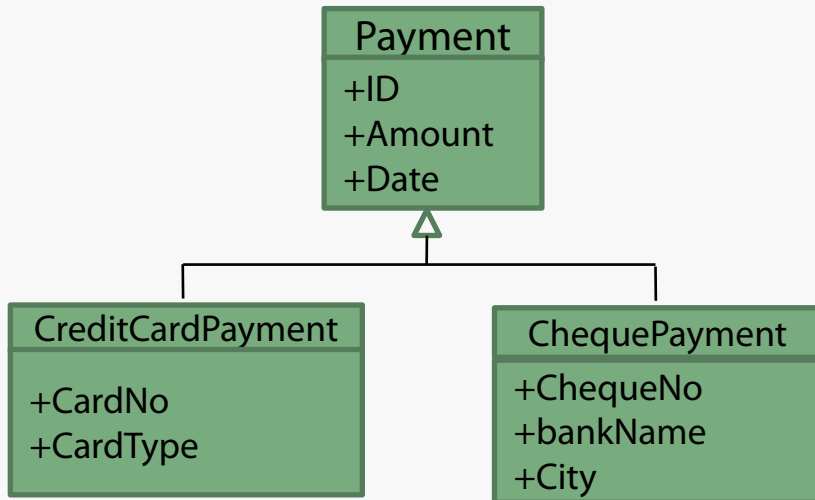
- Object - Relational Mapping
- A persistence framework to map objects to the relational database without writing SQL.
- A programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages.
- ColdFusion ORM – based on Hibernate
  - One of the most mature and popular persistence framework



- ORM Settings
- Mapping a Simple CFC
- CRUD Methods
- Relationships

- Collection Mapping
- Inheritance Mapping
- Embedded Mapping
- Join Mapping
- Using Hibernate mapping files directly

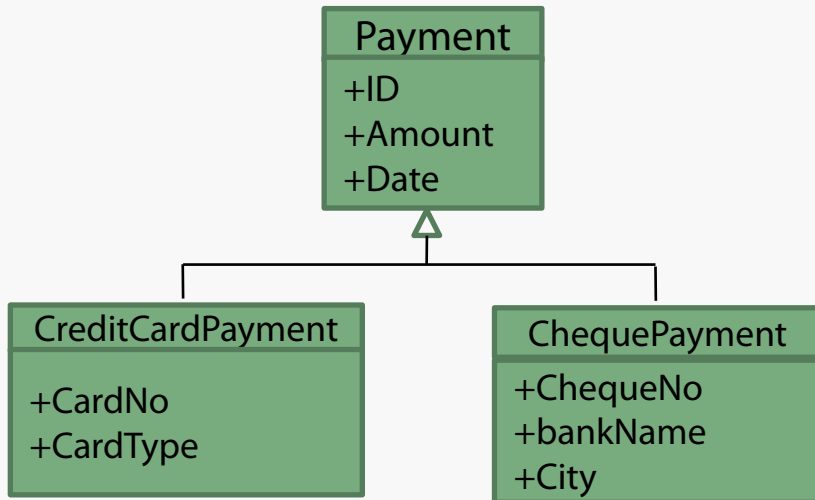
- Three Types
- Table per hierarchy



Payment Table

ID <<PK>>  
Amount  
Date  
PaymentType (discriminator)  
CardNo  
CardType  
ChequeNo  
BankName  
City

- Three Types
- Table per hierarchy
- Table per subclass



Payment Table

ID <<PK>>  
Amount  
Date

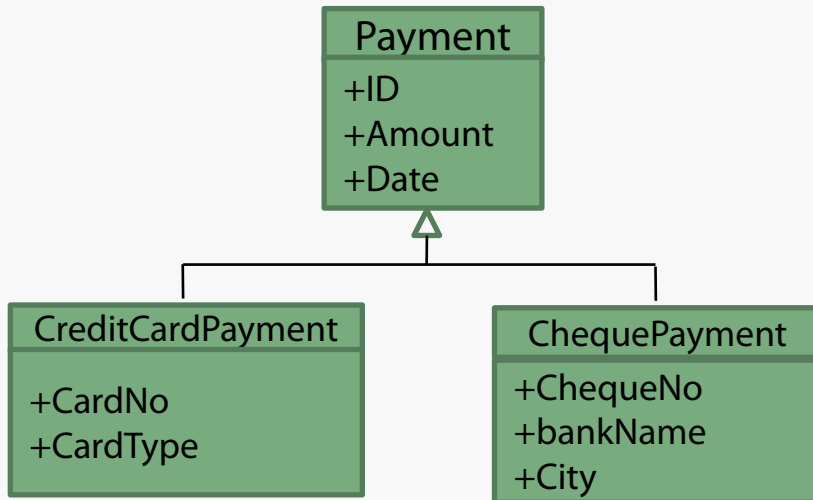
CreditCardPayment Table

PaymentID  
CardNo  
CardType

Cheque Payment Table

PaymentId  
ChequeNo  
BankName  
City

- Three Types
- Table per hierarchy
- Table per subclass
- Table per subclass with discriminator



## Payment Table

ID <<PK>>  
Amount  
Date  
PaymentType (discriminator)

## CreditCardPayment Table

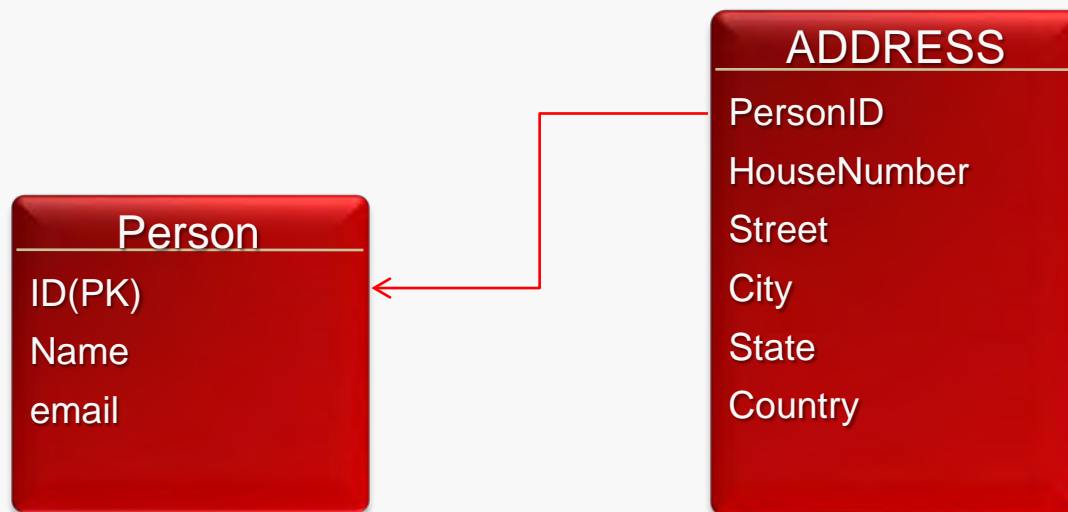
PaymentID  
CardNo  
CardType

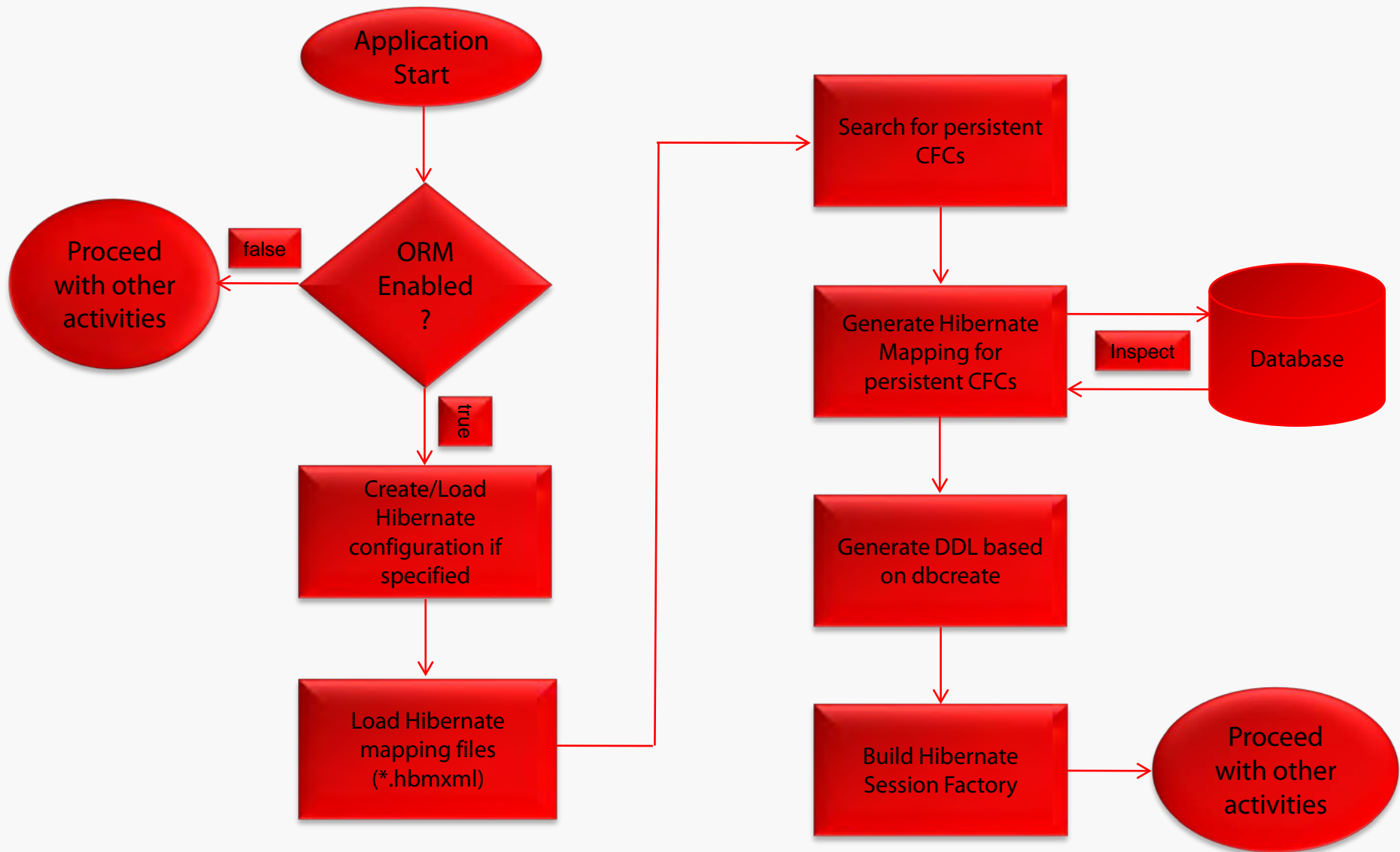
## Cheque Payment Table

PaymentId  
ChequeNo  
BankName  
City

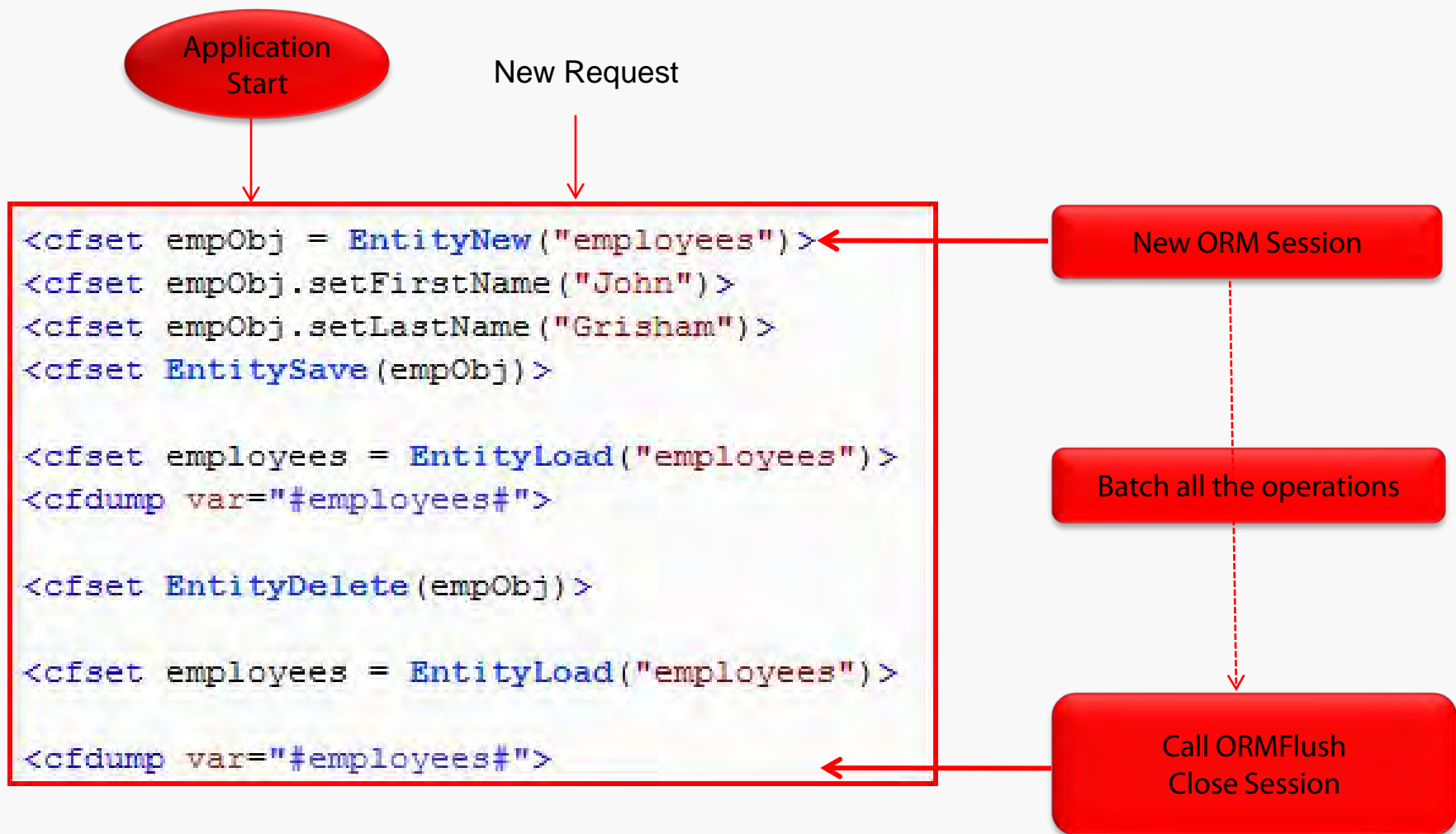
- Useful when using one CFC for multiple tables

```
<cfcomponent persistent="true" table="Person">  
  <cfproperty name="id">  
  <cfproperty name="name">  
  <cfproperty name="city"  
    table="Address" joincolumn="personId">  
  <cfproperty name="country"  
    table="Address" joincolumn="personId">  
</cfcomponent>
```





- Represents a unit of work – typically a transaction
- All the ORM operations happen in a session
- Provides First Level Caching
- Ensures a single instance of an entity.
- Tracks changes made to the objects
- SQLs are executed when session is flushed
  - Typically when the transaction is committed or when the request completes
  - Can use ORMFlush to force
- Automatically managed by CF
  - In most cases, you don't need to worry about it



Application Start

New Request

```
<cfset orderObj = EntityLoad("orders", 5, true)>
<cfdump var="#orderObj#">

<cftransaction action="begin">
  <cfset empObj = EntityNew("employees")>
  <cfset empObj.setFirstName("John")>
  <cfset empObj.setLastName("Grisham")>
  <cfset EntitySave(empObj)>

  <cfset orderObj = EntityLoad("orders", 1, true)>
  <cfset EntityDelete(orderObj)>
</cftransaction>
```

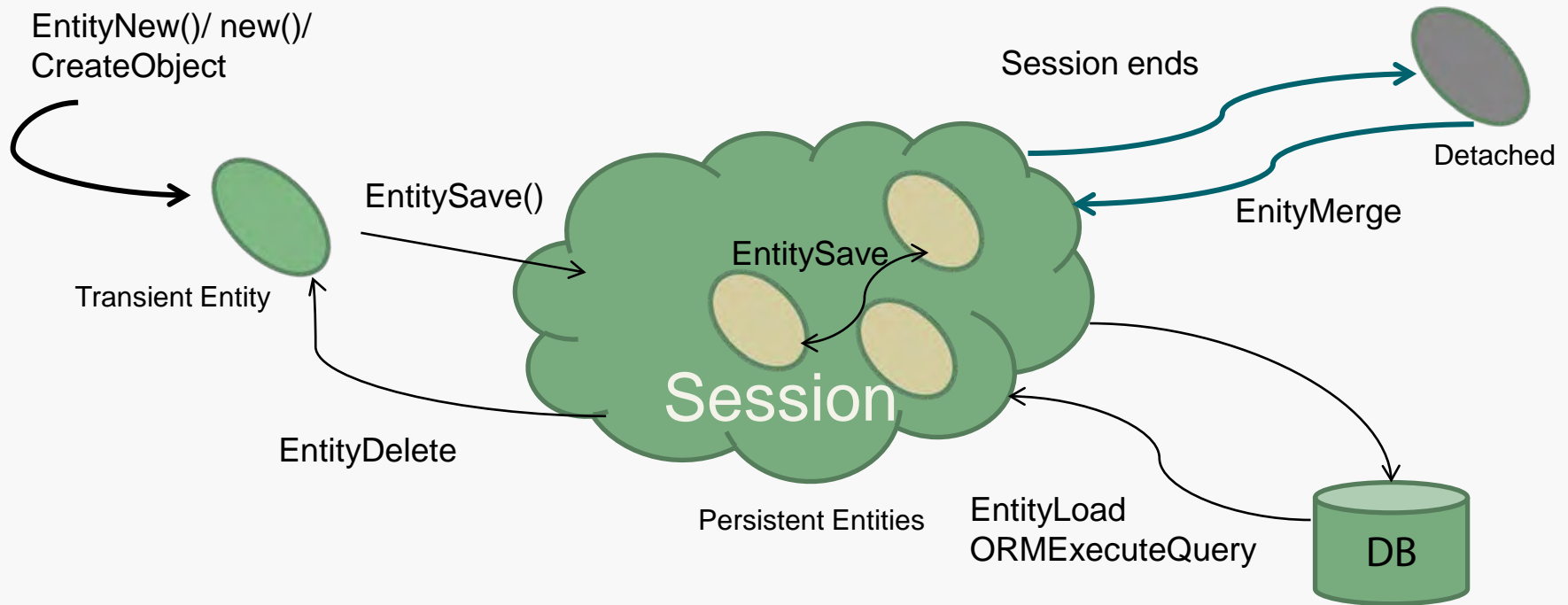
New ORM Session

Call ORMFlush  
Close ORM Session  
New ORM Session

Batch all the operations

Call ORMFlush  
Close ORM Session

- Transient
- Persistent
- Detached



- Optimistic lock for high concurrency
- Update only if the entity is not modified by other thread or externally.
- **optimisticlock** attribute on cfc
  - All
    - All properties are included in where clause of update

```
Update myTbl set col1= newVal, col2= newVal2
                where col1= oldVal and col2= oldVal2
```
  - Dirty
    - Includes only modified fields in the current session
  - Version (default)
    - Checks only version or timestamp column

```
<cfproperty name="lastModified" fieldtype="timestamp|version">
```
  - None
- You can also choose the property for dirty check.

- Immediate fetching

- Fetch target relationship in a separate SQL, immediately

```
<cfproperty name="emp" fieldtype="one-to-many" cfc="order"  
            fkcolumn="EMPID" lazy="false" fetch="select">
```

- Lazy fetching

- Default strategy, `lazy=true`
- On demand, fetch related entities
- `Lazy = "extra"` gets pk of orders and then all order columns from db

- Eager fetching

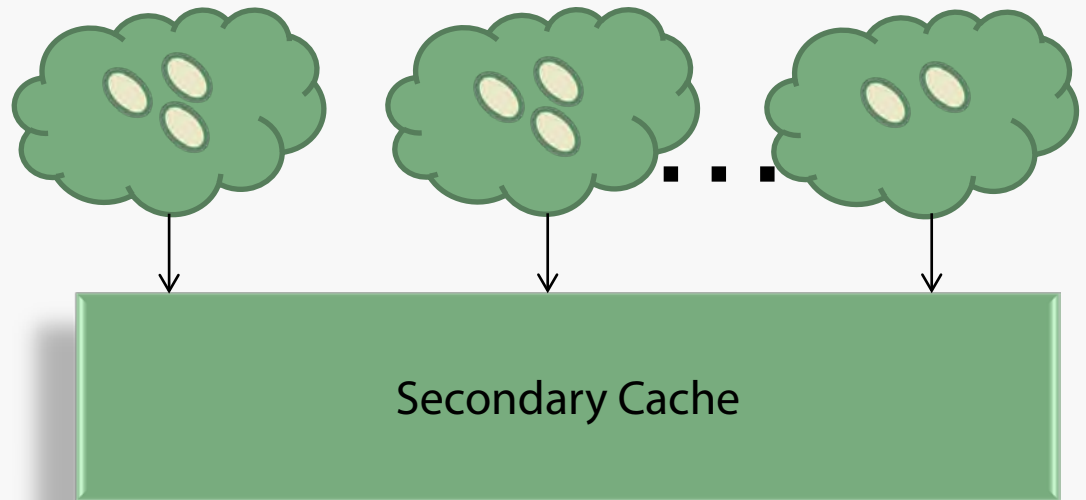
- Fetch together in a single SQL, `fetch="join"`
- Useful for 1:1 frequently used relationships

- Batch fetching

- When fetching relationship, get some more that maybe required later
- Get Addr1 for emp101, plus address for emp102, 103 etc from the table depending on batch size

- **Session Level**
  - Ensures a single instance for a given ID
  - EntityLoad fetches data for the first time
  - Data is cached for the session
  - Next request in the same session will use cached data
  - EntityReload re-fetches
- **Secondary Level Cache**

- Caches data across sessions
- In-memory/disk/clustered
- Pluggable cache impls
- Default - EHCACHE
- Component Caching
- Collection Caching
- Query Caching



- Configuration in Application.cfc
  - ormsettings.secondarycacheenabled
  - ormsettings.Cacheprovider
    - JBossCache, OSCache, SwarmCache, Hashtable, DEFAULT - ehcache
  - ormsettings.cacheconfig
    - Appropriate config file for cache, e.g. ehcache.xml
- In ORM cfc
  - “cacheuse” defines caching strategy
  - “cachename” cache region, a bucket for this data

```
<cfcomponent persistent="true"  
    cachename="foo_region" cacheuse="read-only">
```

- Component

- ```
<cfcomponent persistent="true"  
              cachename="foo_region" cacheuse="read-only">
```

- Relationship

- Primary Key of the associated object is cached
- The associated object itself is cached if coded as above

```
<cfproperty name="arts" fieldtype="one-to-many"  
            cachename="foo_region" cacheuse="read-write">
```

- Query data

```
ORMExecuteQuery("from Art where issold=0", {}, false,  
                {cacheable=true, cachename="foo_region"});
```

- Read-only
  - Best performance for read only data
- Nonrestrict-read-write
  - Use when data is updated occasionally
- Read-write
  - Use if your data needs to be updated
  - More overhead than the two preceding strategies
- Transactional
  - Transactional cache
  - Can only be used if the cache provider is transaction aware

- ORMEvictEntity

```
ORMEvictEntity("<component_name>", [primarykey])
```

- ORMEvictCollection

```
ORMEvictCollection("<component_name>", "<relation_name>",  
[primarykey])
```

- ORMEvictQueries

```
ORMEvictQueries([cachename])
```

- Set `ormsettings.eventhandling="true"`
- CFC level
  - `preinsert` and `postinsert`
  - `predelete` and `postdelete`
  - `preupdate` and `postupdate`
  - `preload` and `postload`
- Application Level
  - Set `ormsettings.eventhandler="AppEventHandler.cfc"`
  - Should implement the `CFIDE.orm.IEventHandler` interface

- Tables created on Application startup
- **ormsettings.dbcreate**
  - Update – create new or update if table exists
  - Dropcreate – drop and then create table
  - None – do nothing
- **ormsettings.sqlscript**
  - Executes the sql script at the time of table creation.

- Defines the strategy for naming tables and columns
- **ormsettings.namingStrategy**
  - Default – CFC/property name matches table/column name
  - Smart – camel cased name is uppercased with “\_” between words.
    - CFC “OrderProduct” is “ORDER\_PRODUCT”, OrderID is “ORDER\_ID”
  - Your own CFC that implements **cfide.orm.INamingStrategy**

```
Component UCaseStrategy implements cfide.orm.INamingStrategy
{
    public string function getTableName(string tableName)
    {
        return Ucase(tableName);
    }
    public string function getColumnName(string columnName)
    {
        return Ucase(columnName);
    }
}
```

# Q & A



# Thank You !

- [rukumar@adobe.com](mailto:rukumar@adobe.com)
- <http://www.rupeshk.org>

