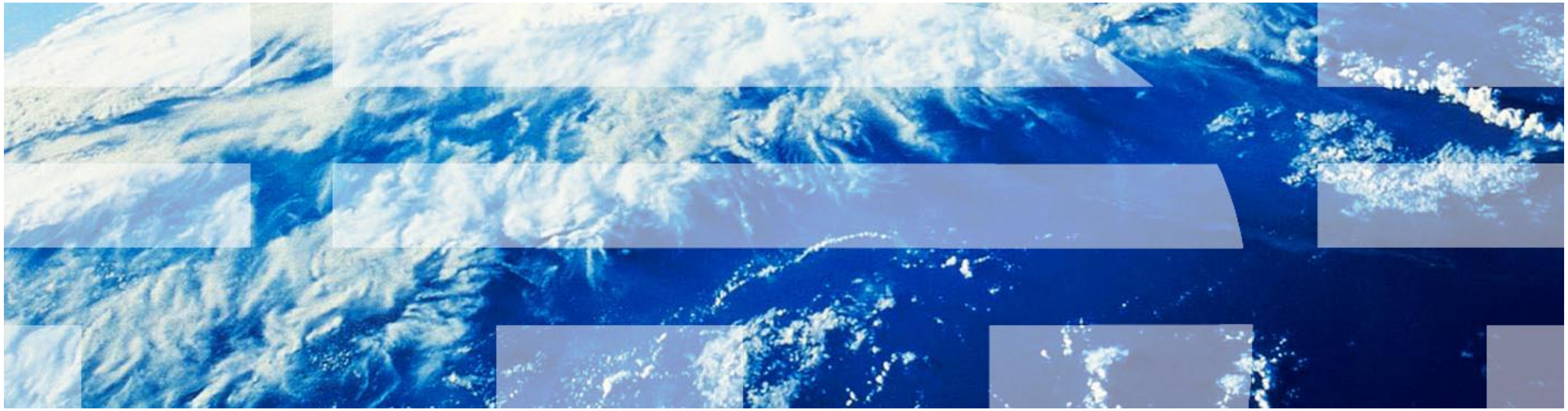
Customer POC Experience with WebSphere eXtreme Scale

eXtreme Scale caching alternatives for Bank ATM Offerings

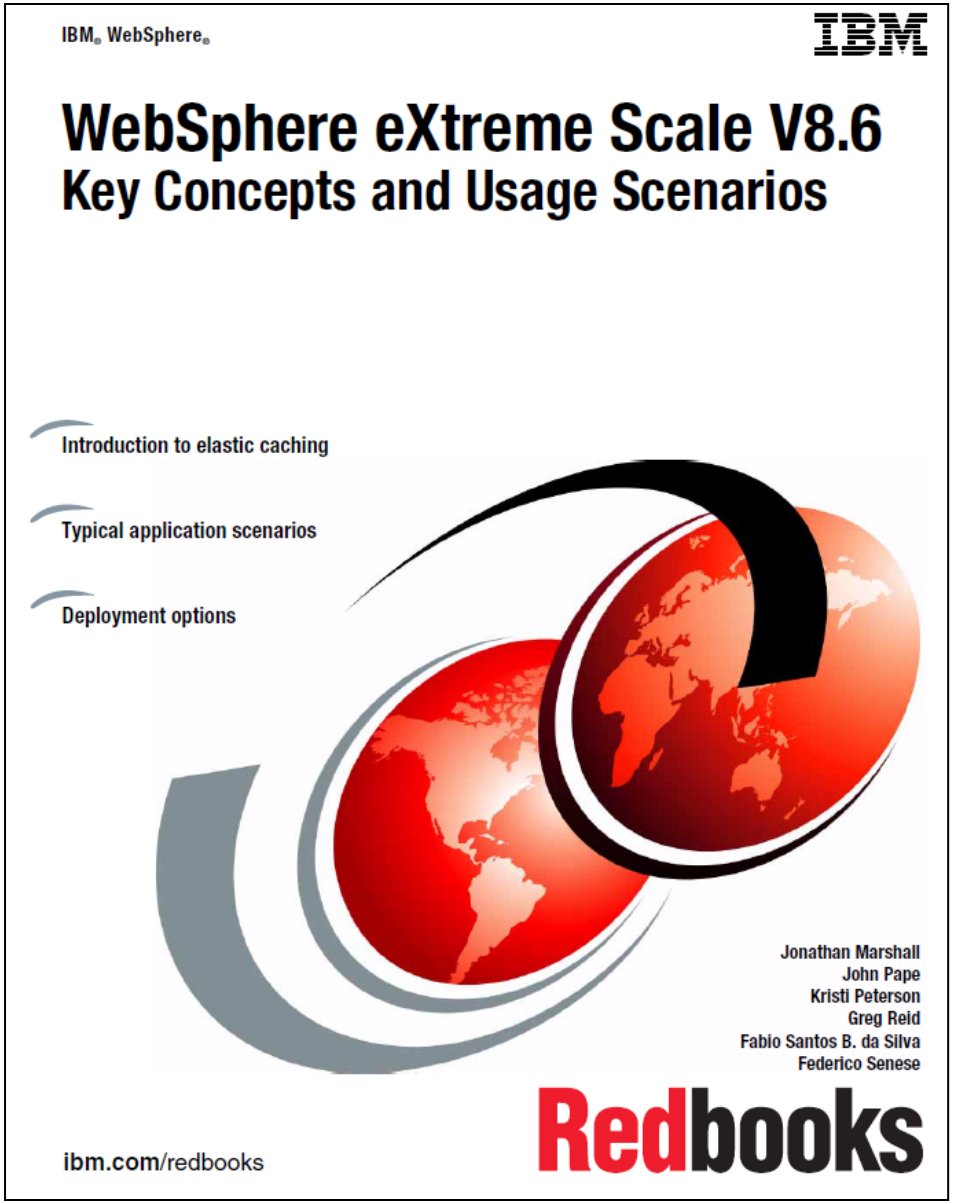


Agenda

- Business and application challenges where elastic caching applies
- Customer POC Context and Goals
- Customer POC Scenario
- Problem Statement
- Solution Requirements
- Solutions and Alternatives for consideration

- Demonstration of one of the alternatives
  - In-line cache
    - Write-Behind loader to backend database
  - Pre-load cache from database

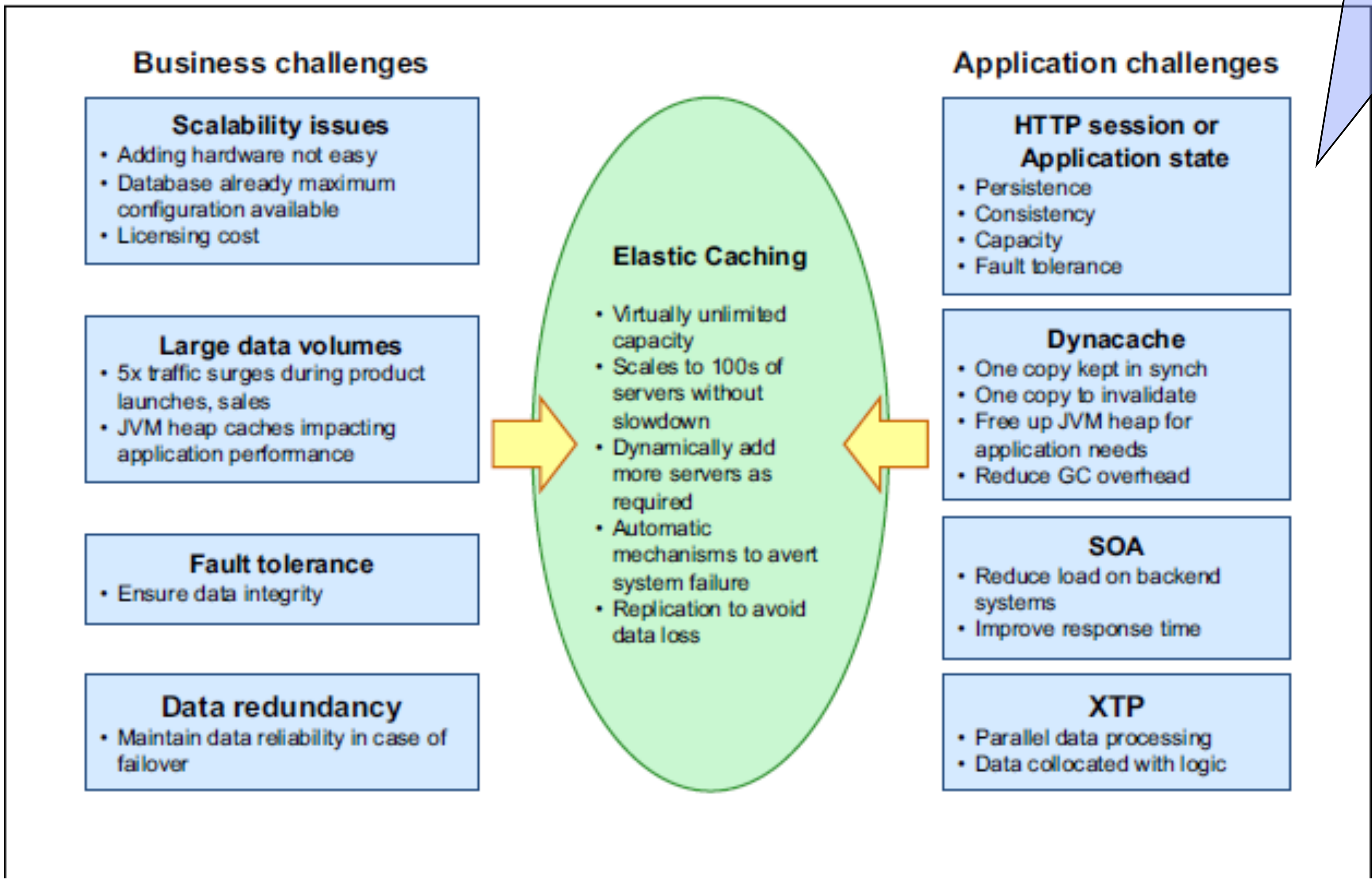
WebSphere eXtreme Scale V8.6 Redbook



Business and Application challenges

WXS elastic caching offers solutions for various business and application challenges

- Http Session Cache
- Application side cache
- Application Inline cache



Business and Application challenges

- The **application state** store and **side cache** scenarios are typically simple to implement and are widely used

- The **in-line cache** and **Extreme Transaction Processing** scenarios are more advanced use cases
  - Typically involves some code running on the grid itself
    - Agents / loaders, etc
  - These scenarios apply only to WebSphere eXtreme Scale, and not to the WebSphere DataPower XC10 Appliance

Scenario	Complexity	Application change required	System of record
Application state store	Simple	Depends	Grid
Side cache	Simple	Depends	Back end
In-line cache	Medium	Yes	Grid
Extreme transaction processing	Medium to advanced	Yes	Grid

Context and Goals

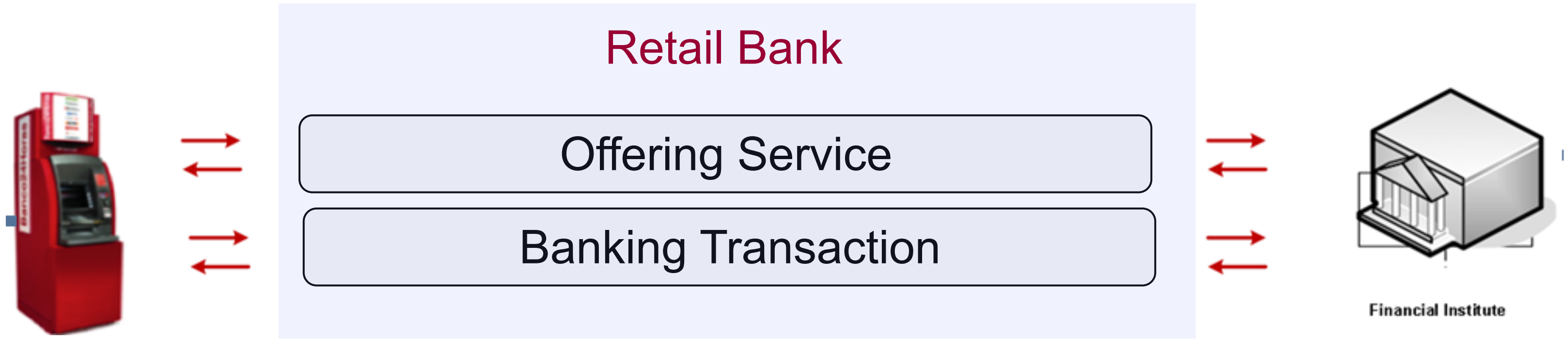
Context

- Customer is **modernizing** its solution of **Transaction Offerings** for ATM banking services
- The existing **proprietary** caching architecture is difficult to maintain
- Existing cache solution is not sharable between applications and runs in the application JVM.

Goal

- The **new solution** will employ the mechanims of a **distributed cache** using **IBM eXtreme Scale** for improved performance, availability and maintenability of the offerings

Retail Bank - Scenario



1. User inserts ATM card from their financial institution into the ATM
2. Offering Service at "Retail Bank" reads card information and contacts the "Financial Institution"
   - Obtains the specific "Offerings" available for the card holder on the account
     - Such as: Withdrawal, funds transfer, acct balance, etc
3. Offering Service applies the "Offering Parameters" required by the financial institution
   - Such as maximum daily withdrawal limit
4. Offering Service renders the offerings to the ATM user
5. From the ATM, the user initiates one of the available banking transactions from the list of offerings presented, as obtained from the card holders financial institution

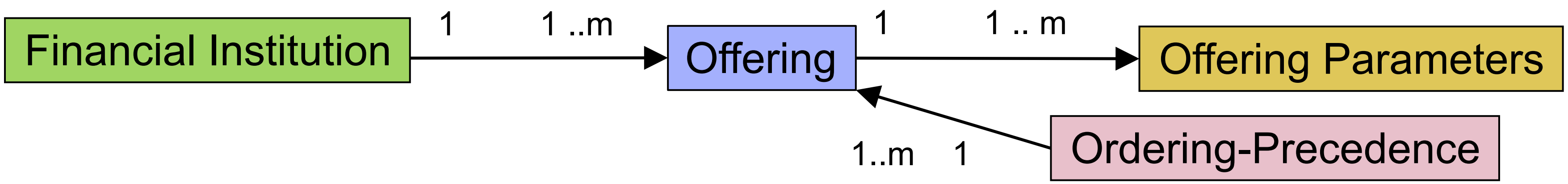
- Data to be cached in the Banking ATM Offering Scenario
  1. Financial Institution data
  2. Offering data
  3. Offering Parameter data

Current Architecture

- Data is configuration tables from a back-end database
  - Basic Information used for the business rules
  - Offering application only “reads” the data
  - Low refresh rate (Less that 1 time per day)
    - Outside applications are used to update the database tables via web application as changes are required, such as:
      - Offering Parameter changes
      - Offering is added or removed from a financial Institution
- Existing Cached data size is relatively small, But……….
  - We will discuss the problem statement

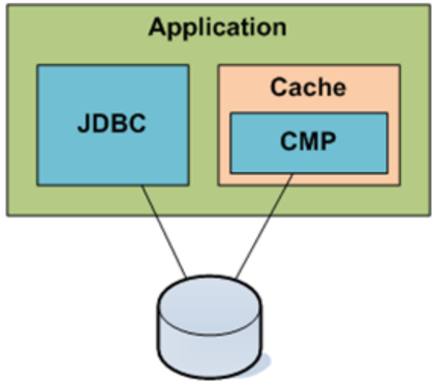
<u>Data Mapping</u>

- Each Financial Institution has many Offerings
- Each Offering can have one to many Offering parameters
- Ordering Precedence data is used to appropriately order query results



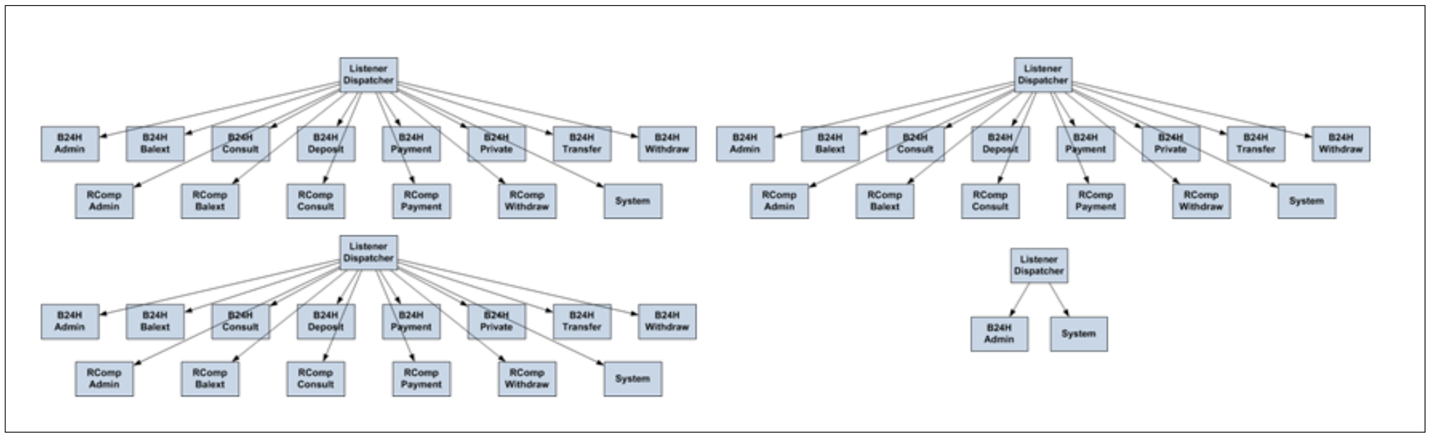
Problem Statement

- Proprietary cache implementation
  - Local cache in every application space (Shares JVM heap with application)
  - Cache logic is difficult to maintain



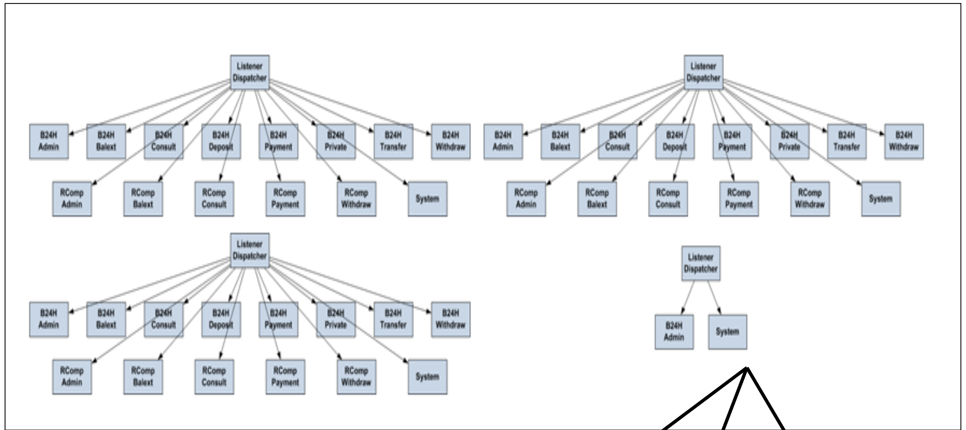
- Local cache in every application space
  - No cache sharing
  - There are 15 applications, each with its own copy of the cache data
  - Example: 40 MB of cache is now 40 MB * 15 Applications = 600 MB

- Application Architecture requires three (3) variants of each of the 15 applications based on routing to specific financial institutions
  - Example: 40 MB cache is now 600 Mb * 3 variants of application = 1.8 GB

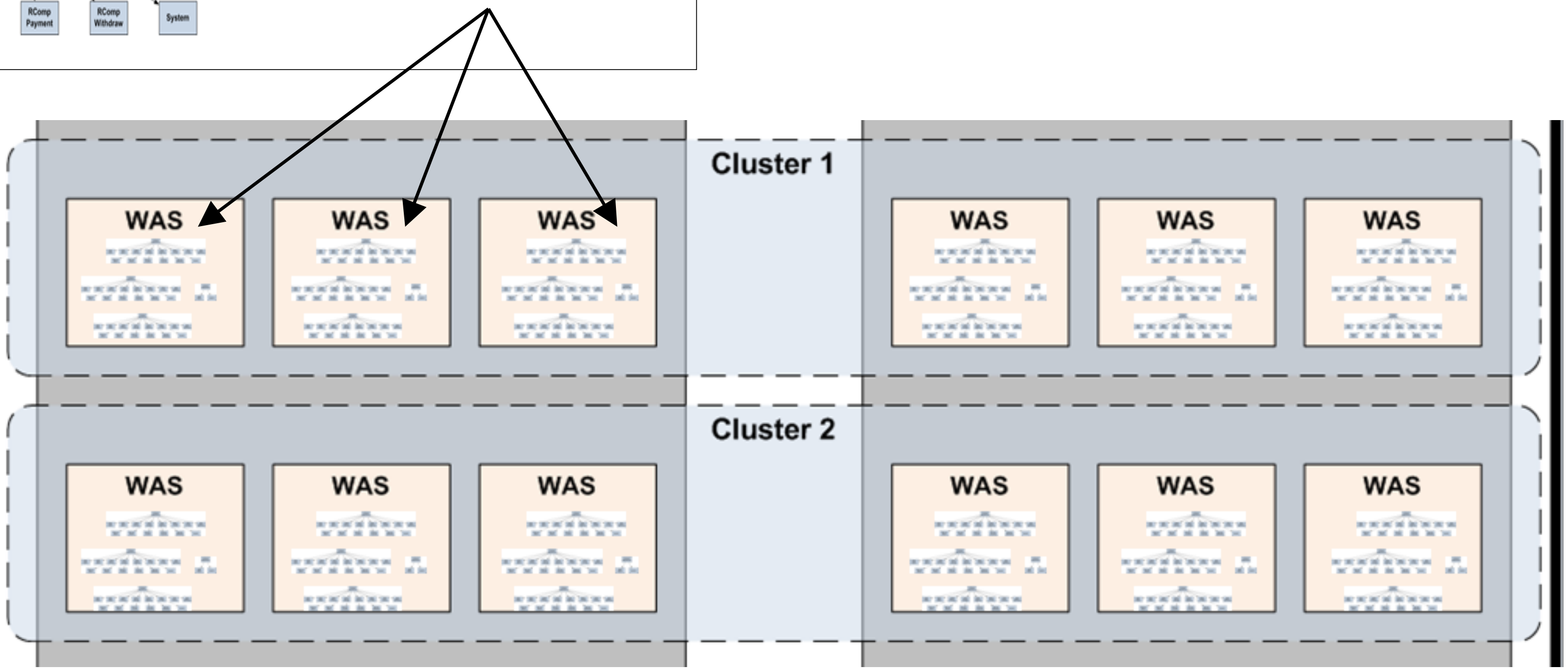


Problem Statement

- 12 WebSphere Servers supporting the application
  - Each server contains the 1.8 GB of cache
  - Example: 40 MB cache is now 1.8 GB * 12 JVms = 12.6 GB



- Cache refresh is performed through the reading of all records of the entity that has changed
  - Updating all of the caches in each JVM results in minutes of unavailability



Solution Requirements

- The adoption of a distributed cache will be designed taking into consideration the following requirements:
  - Coding simplification of the cache implementation
    - Compared to existing proprietary cache implementation
  - Cache sharing between applications and JVMs
  - Reliability and security for cache operations (insert, update, and delete, and queries)
  - Immediate availability of the cache to applications, even if applications or application servers are restarted

Solution and alternatives

- **Option 1:** Side Cache using ObjectMap API

  Pros:

  - Fairly easy to implement
  - Code changes are not exhaustive and typically contained in the Data Access layer of the application
  - Cache preloading can be straight forward and efficient
  - Application can continue to function if the cache is out of service
  - Solution could also be implemented using DataPower XC10 appliance

  Cons

  - Does not support synchronization of cache changes to DB at runtime

Offload Redundant Processing : Side Cache

1. Applications check to see if WebSphere eXtreme Scale contains the desired data.
2. If the data **is there**, the data is returned to the caller.
3. If the data is **not there**, the data is retrieved from the back-end
4. Insert the data into WebSphere eXtreme Scale so that the next request can use the cached copy.

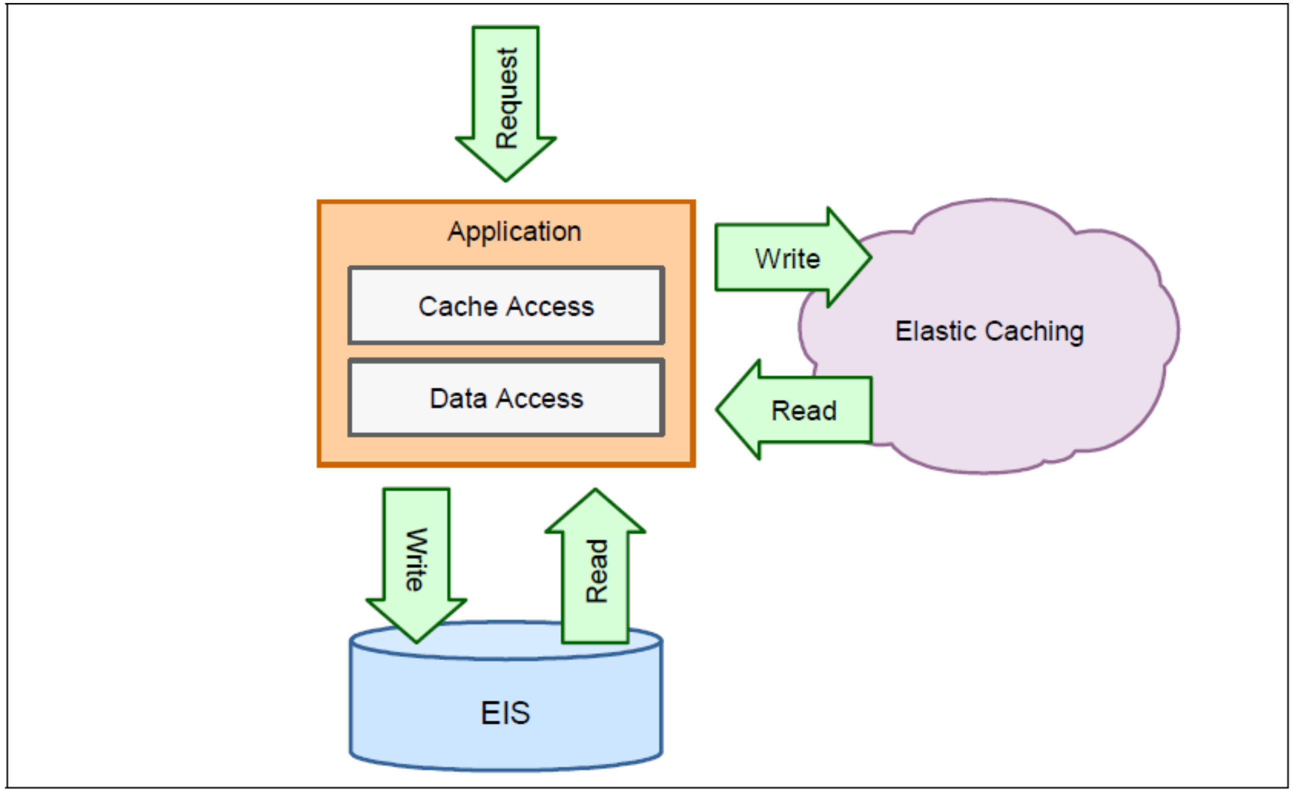


Figure 3-2 Side cache scenario

Solution and alternatives

- **Option 2:** In-Line Cache and (Optional) extreme processing for queries and WXS agents

<u>Pros:</u>

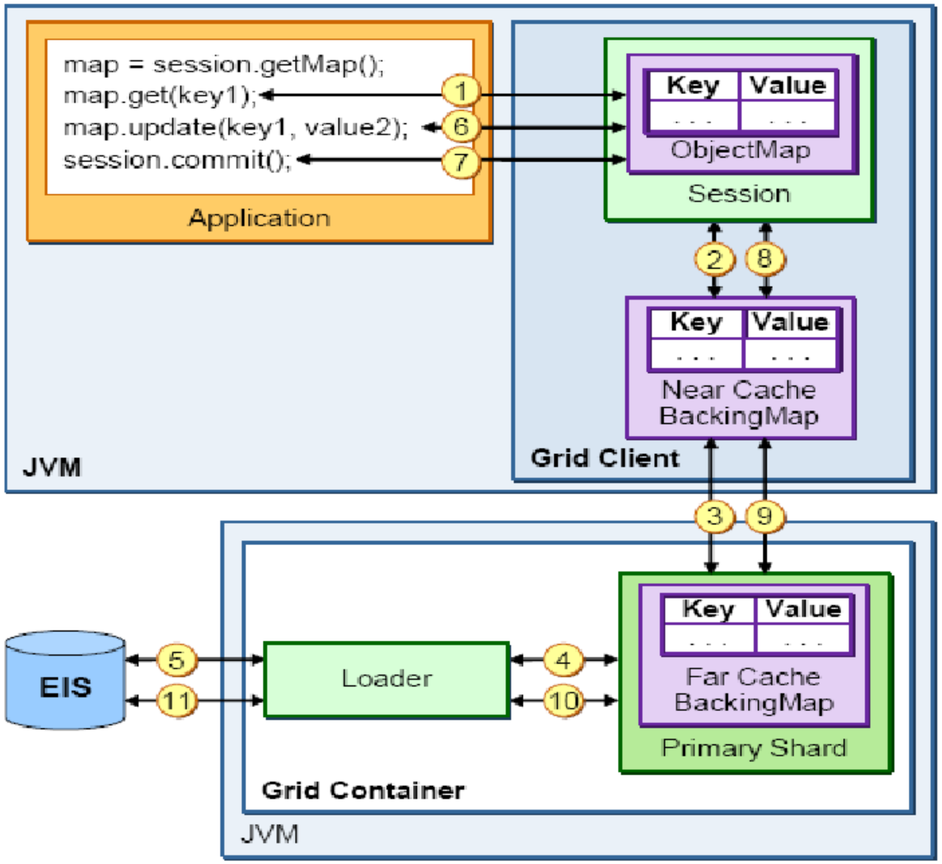
- Custom loaders can be developed to keep the database synchronized with cache updates
- Option to preload the cache upon initialization of the backing maps
- (Optional) Agents can be developed for parallel query and cache updates across partitions

<u>Cons</u>

- Longer term investment than the Side cache scenario
- Additional development time required build the WXS solution with WXS agents and loader plug-ins
- Application must use WXS APIs to interact with the cache as the “Loader” is responsible for interaction with the database
- Solution will only run on Extreme Scale, not Datapower XC10

Offload Redundant Processing : In-line cache with Write-Behind

- Changes are written to the back-end **asynchronously**
  – A *write-behind* cache
- Back-end load is significantly reduced as there are fewer but larger transactions
  – WXS configuration is used to determine when to perform the batch updates to the database, based on elapsed time / number of transactions since last buffering period
- Back-end availability has no impact on application availability

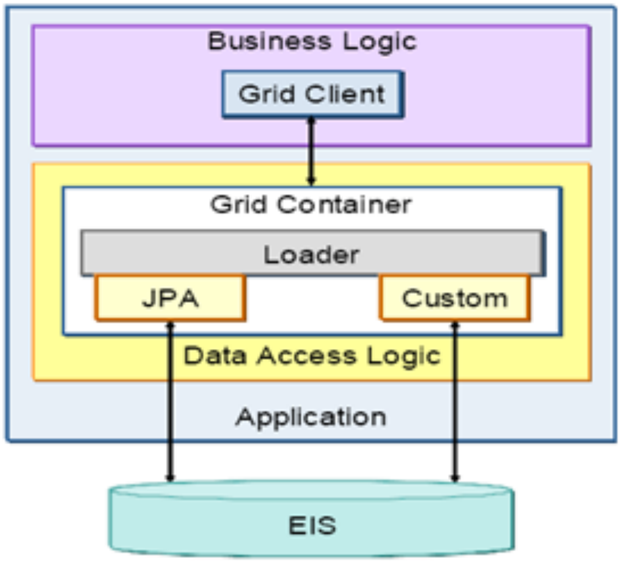


Application logic:

1. Look in WXS cache for object
2. Work with the object that was returned from the WSX grid
   - Data could have come from the cache or the database, WXS loader takes care of that
   - The client does not interact with the database!

Inline Cache

- Whether the data is in the cache or not becomes transparent to the application
  - the application sees an extremely fast back-end access (assuming the cache is large enough to provide a good hit rate
- An inline cache requires the implementation of a “Loader”
  - Prebuilt “JPA” loaders ship with WXS
  - Or develop your own “Custom” loader using the WXS **Loader** interface



Example Solution

- **In-line cache scenario** using custom JDBC loader plug-in
  - Requirement: Customer would like to allow real-time cache updates and have the cache updates synchronized to the backend database
  - Currently, updates are scheduled each evening and the cache I reloaded

- **Pre-load the cache** with all of the data from the database
  - Requirements:
      - The application performance cannot tolerate database access latency on cache misses

- **Load the cache using data de-normalization and data partitioning** based on usage patterns in the applications
  - Requirements:
    - Efficient access of the cached data in the IMDG (In Memory data grid)
      - Efficient use of the WXS query API can be accomplished if related graph data is co-located into the same partition as it's root object
      - More efficient queries can be developed if certain data from database tables are de-normalized to avoid expensive "join" operations
      - Can implement cache queries using a single partition query to support the use case scenarios

Example: POC primary queries to support

- Query all **Offerings** associated with the **Financial Entity,** ordered by **precedence**

```
SELECT  o.id,  o.name,  o.active,   o.wheretypeid,    o.whotypeid,   o.financialentityid,
            wo.precedence "whoPrecedence",
            we.precedence "wherePrecedence"
FROM   T_OFFERING o,  T_WHOTYPE  wo,   T_WHERETYPE  we
WHERE       o.wheretypeid = we.id
AND          o.whotypeid = wo.id
AND          o.financialentityid = ?
ORDER BY  wo.precedence,  we.precedence
```

ID	NAME	ACTIVE	WHERETYPEID	WHOTYPEID	FINANCIALENTITYID	whoPrecedence	wherePrecedence
1	Offering 1	1	1	3	111	2	0
2	Offering 2	1	2	2	111	1	1

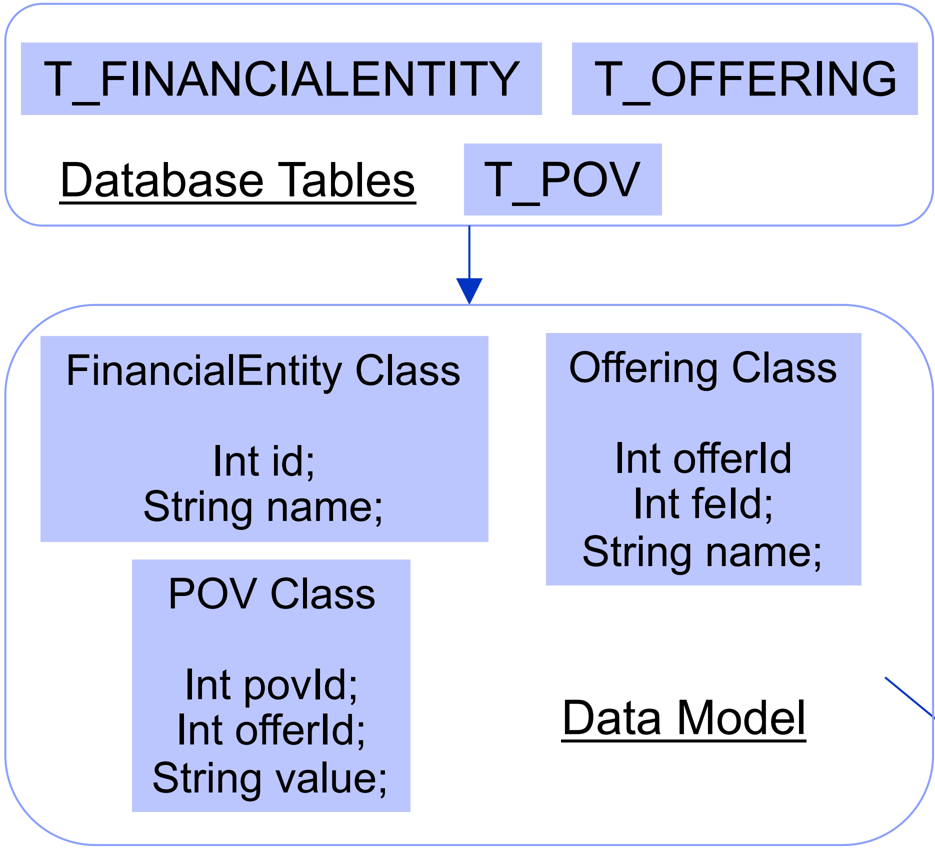
- **<u>Example</u>** above shows simple data denormaliztion option
  - Combine data from the Offering, Whotype and WhereType tables into a single cacheable object
  - Eliminates the need to use 'joins' in the WXS GRID query to obtain the cached data
- Precedence columns used specifically to order the query ResultSet by precedence
- Two very small tables containing 3 or 4 rows

- What other technique could be used to load the grid to EFFICIENTLY query the cached data using ORDER BY o.whoType.precedence, o.whereType.precedence? Explore next!

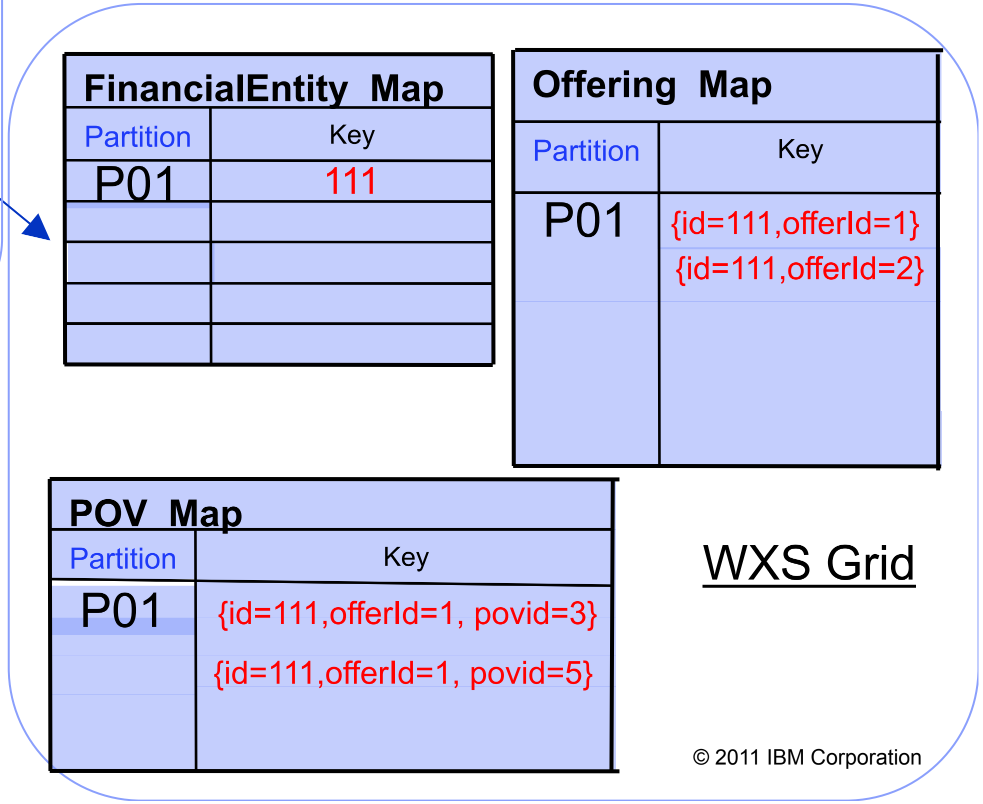
WXS Data modeling options

- Define a viable data model
  - A partitioned environment requires special considerations for holding object graphs
  - **Should the entire object graph instance be held in one partition?**
  - Considerations include:
    - How imbalanced will the grid partitions become?
    - How can we place data into specific partitions based on a parent object?
  - **Should the data be denormalized to combine fields from child database tables into a single WXS model object?**
  - Considerations include:
    - Size of the grid will grow faster. Is that an issue?
    - How does the data get properly updated back to the database?
  - **Should some Objects just be loaded into all partitions?**
  - Considerations include:
    - How many records of data need to be stored in each partition?
    - Could be good solution if the size is small, say storing US states, or the WhoType and WhereType data from our example POC scenario
    - How can you duplicate data into all partitions?
    - What does this mean for partition routing to access the data?

Collocate Master and child objects in the same partition



- Co-locate Master and child objects in same partition
  - FinancialEntity key = id
  - Offering Key is a composite POJO that includes the FinancialEntity key “id” and the Offering key “offerId”
  - POV Key is a composite POJO that includes the OfferingKey POJO and the POV key “povId”



- Requires custom POJOs to be developed as the Key Classes for Offering and POV
  - Implements the WXS PartitionsableKey interface
  - Generates a HASH for the Offering and POV identical to the Master “FinancialEntity” object

Collocate Master and child objects in the same partition

OfferingKey POJO class

- Implements PartitionableKey from WXS
- WXS calls the ibmGetPartition() method
  - It returns the integer value of the Financial Entity
  - So the Offering will hash to the same value as its parent Financial Entity

OfferingKey Code example

```
public class TOfferingKey implements PartitionableKey {

  int offringKey;
  int financialEntityKey;

  public TOfferingKey(int offringKey, int financialEntityKey) {

                super();
                this.offringKey = offringKey;
                this.financialEntityKey = financialEntityKey;
  }

public Object ibmGetPartition() {
                return Integer.valueOf(financialEntityKey);
}
```

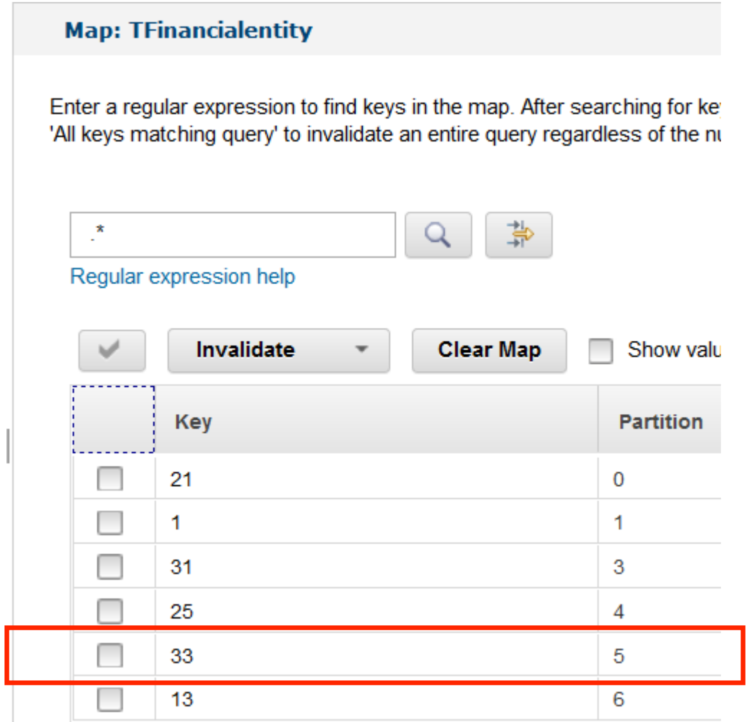
- In the Loader code, create a new instance of the TOfferingKey
  - Pass in the id of the Financial Entity and the id of the Offering on the constructor
- Put the entry in the WXS map using the hashed Key

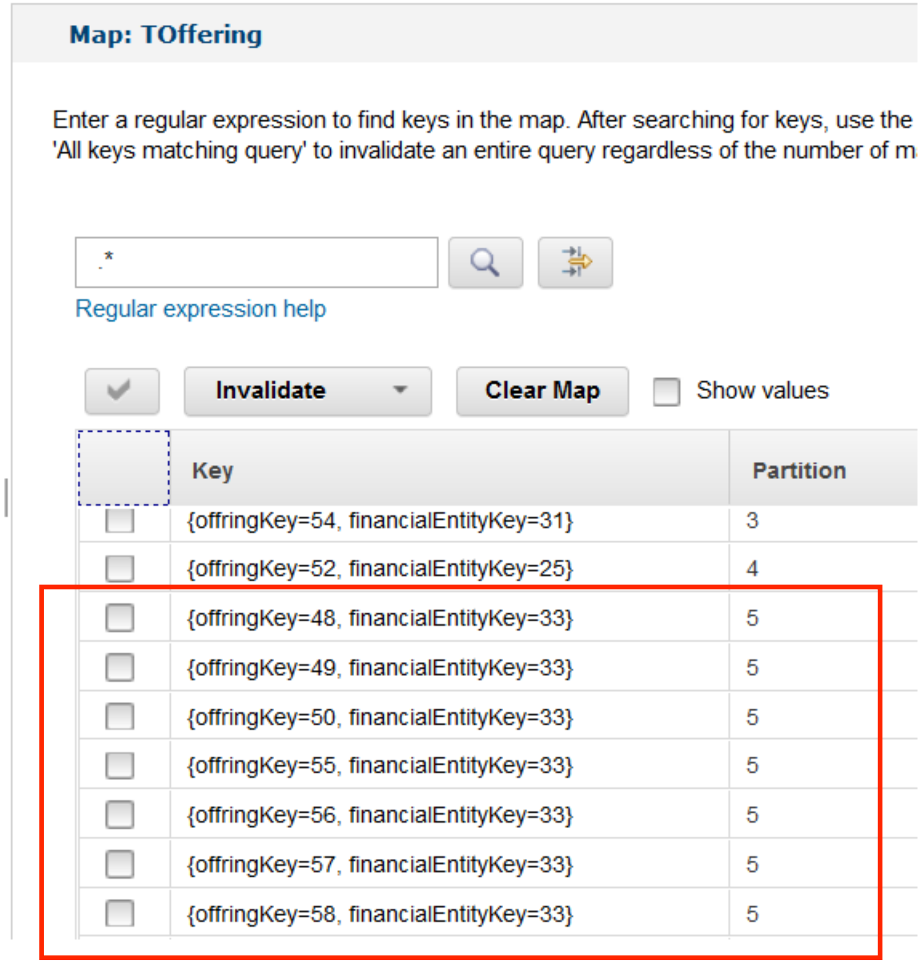
Loader Code example

```
TOfferingKey offeringKey = new
TOfferingKey(offeringid, feid);


map.put(offeringKey, toffering);
```

Graph of objects loaded into the same partition for efficient single partition query

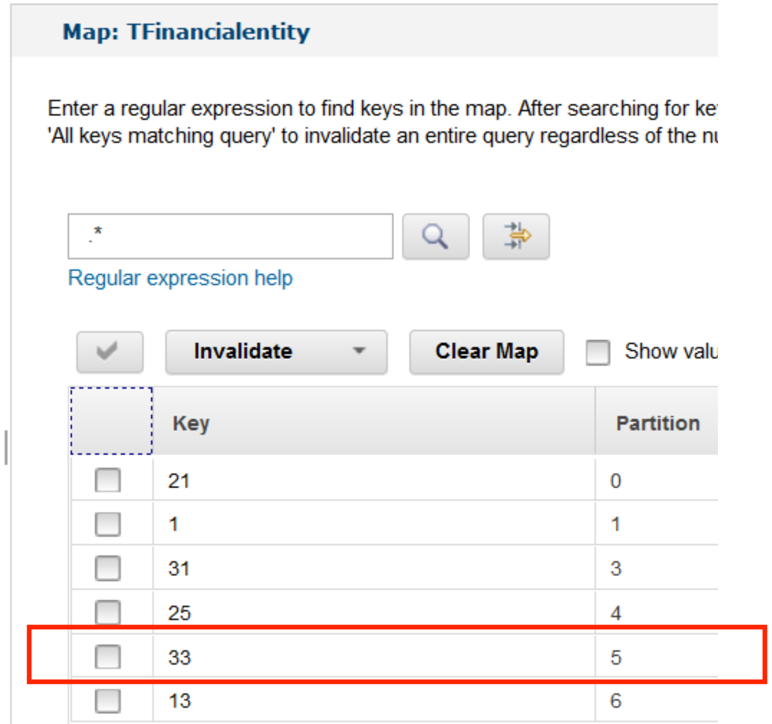


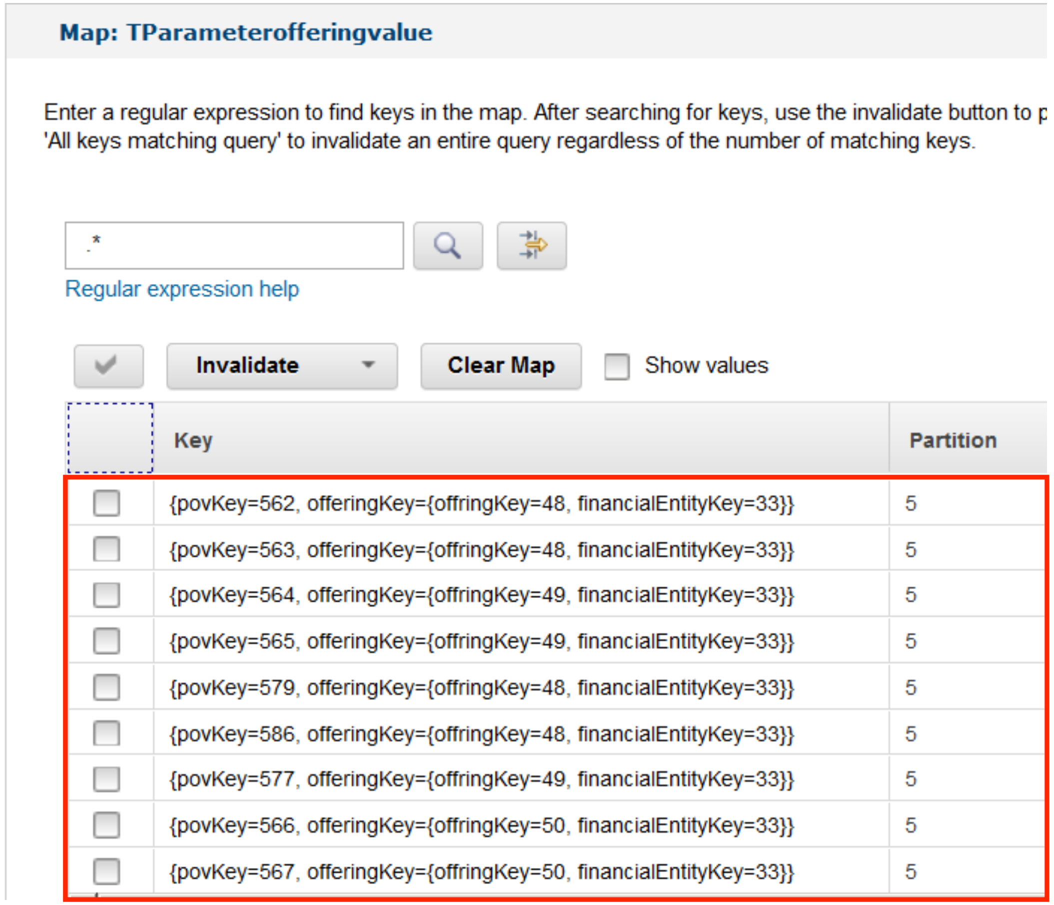


SELECT o FROM TOffering o WHERE o.feld = ?1

The query can be executed in a single partition to obtain ALL related offerings for a given financial institution

Graph of objects loaded into the same partition for efficient single partition query





SELECT p FROM TParameterOfferingValue p WHERE p.offeringid = ?1

The POV query can be executed in a single partition to obtain ALL related POVs for a given Offering

Denormalize the data model

```java
public class TOffering implements Serializable {

    private static final long serialVersionUID = 1L;

    private int feId;

    private int offerId;

    private String name;

    private String active;

    private int wheretypeid;

    private int whotypeid;
```

```java
public class TWhotype implements Serializable {
    private static final long serialVersionUID = 1L;

    private int id;

    private String label;

    private String name;

    private int precedence;
```

- **Original Offering class**
- Fields map to the columns in the database table

- For queries of child objects to work, they must be in the same partition
- For queries that use ORDER BY to work, the objects used for the ordering must be in the same partition
- An implicit JOIN is incurred to obtain the whotype.precedence and wheretype.precedence

  - Example SQL: Similar WXS Query is required to obtain from the cache:

```sql
SELECT  ............
FROM   T_OFFERING o,  T_WHOTYPE  wo,
    T_WHERETYPE  we
WHERE       o.wheretypeid = we.id
AND         o.whotypeid = wo.id
AND         o.financialentityid = ?
ORDER BY  wo.precedence,  we.precedence
```

Denormalize the data model

```
public class TOffering implements Serializable {
    private static final long serialVersionUID = 1L;
    private int feId;
    private int offerId;
    private String name;
    private String active;
    private int wheretypeid;
    private int whotypeid;
```

```
public class TOffering implements Serializable {
    private static final long serialVersionUID = 1L;
    private TOfferingKey tofferingKey;
    private int povId;
    private int feId;
    private int offerId;
    private String name;
    private int whoTypePrecedence;
    private int whereTypePrecedence;
```

- **New Offering class that has been denormalized**
- The Offering class contains fields for the **precedence** of the whotype and wheretype

Considerations:

- The WhoType and WhereType objects no longer need to be in any specific partition
- When pre-loading the grid, or in a side cache scenario, an appropriate SQL query to pull the data from the DB to be placed into the Offering class when added to the cache.
- Let's see how this SQL looks (From preloader)

SELECT O.*,P.ID AS POVID, wheret.precedence as WHEREP, whot.precedence as WHOP FROM KEVINLP.T_OFFERING O, KEVINLP.T_PARAMETEROFFERINGVALUE P, T_WHERETYPE wheret, T_WHOTYPE whot WHERE O.ID = P.OFFERINGID and O.wheretypeid=wheret.id and O.whotypeid=whot.id

ID	NAME	ACTIVE	FINANCIALENTITYID	POVID	WHEREP	WHOP
62	OS_0001_IF_...	1	1	657	3	2
62	OS_0001_IF_...	1	1	658	3	2
62	OS_0001_IF_...	1	1	659	3	2
62	OS_0001_IF_...	1	1	660	3	2

Denormalize the data model

```java
public class TOffering implements Serializable {

    private static final long serialVersionUID = 1L;

    private TOfferingKey tofferingKey;

    private int povId;

    private int feId;

    private int offerId;

    private String name;

    private int whoTypePrecedence;

    private int whereTypePrecedence;
```

Retrieving the data from the Offering object

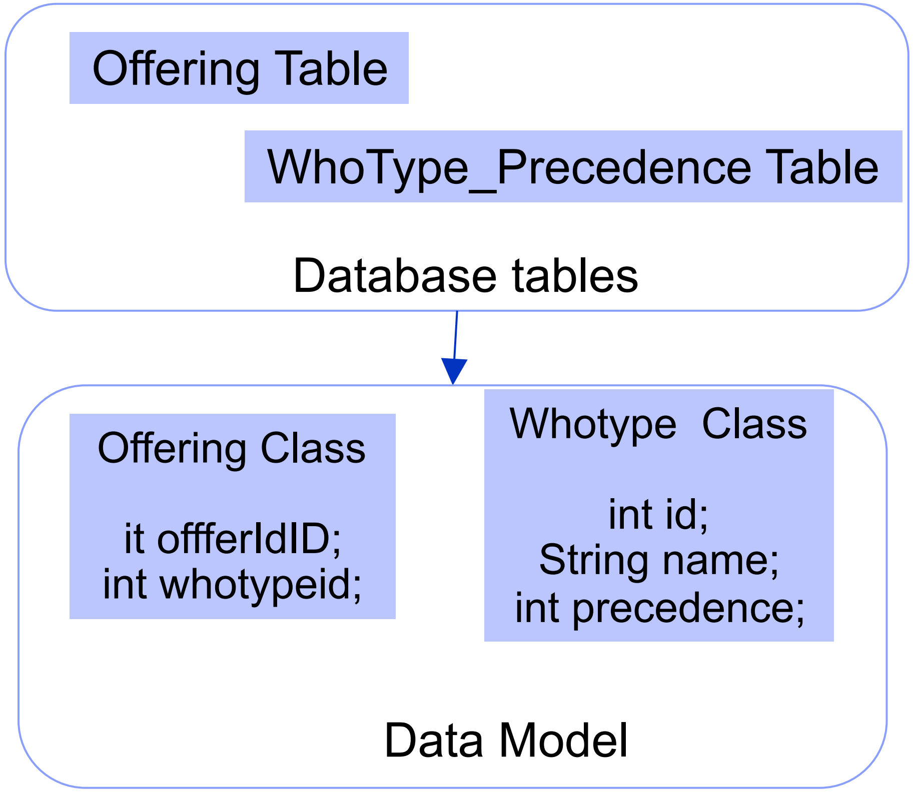
- The new fields can be retrieved from the Offering class as shown below using the getter() methods on the Offering class.
- Or used in an ORDER BY Clause in a WXS query
- ** If the query required to run on multiple partitions, the client must order (Sort) the results form the partitions.

- Another alternative is to load the Whotype and Wheretype objects into ALL partitions in the grid
  - Then the queries that use these objects will work from any partition

```java
for (TOffering toffering : tofferings) {
    //System.out.println("FE ID = " + toffering.getFeId());
    //System.out.println("POV ID = " + toffering.getPovId());
    System.out.println("OFFERING ID = " + toffering.getOfferId());
    System.out.println("OFFERING Name = " + toffering.getName());

    //TODO KLP added to get the precedence of the whotype and wheretype of teh offering
    System.out.println("OFFERING Whotype Precedence = " + toffering.getWhoTypePrecedence());
    System.out.println("OFFERING Wheretype Precedence = " + toffering.getWhereTypePrecedence());
```

Example: Duplicate data into all partitions (MapGridAgent)



- **Duplicate data into all partitions**
  - This technique works best for small sets of data
  - Query of **Offerings** where the ordering is required by the "**Whotype precedence**" will now work in any partition

FinancialEntity Map

Partition	Key
P01	111

Offering Map

Partition	Key
P01	{id=111,offerId=2}

- Loading data into all partitions can be accomplished by:
  - Run a query to fetch all of the Whotype rows from the database
  - Develop a WXS Agent to insert them into each partition.
  - Agents run in every partition, so this task is quite simple

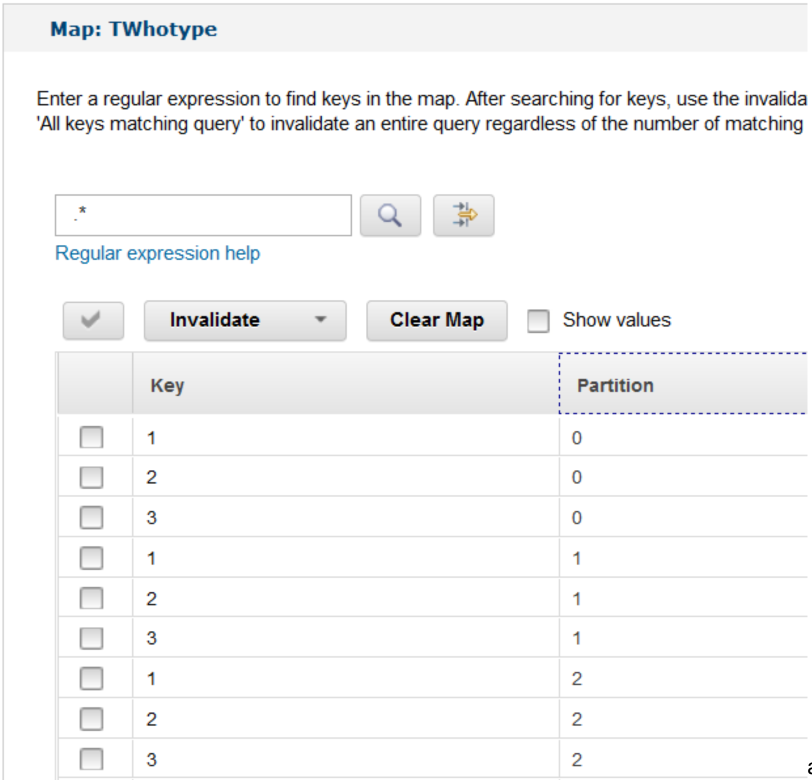
WhoType Map

Partition	Key
P01	1 4
P02	1 4

WhoType loaded into ALL partitions

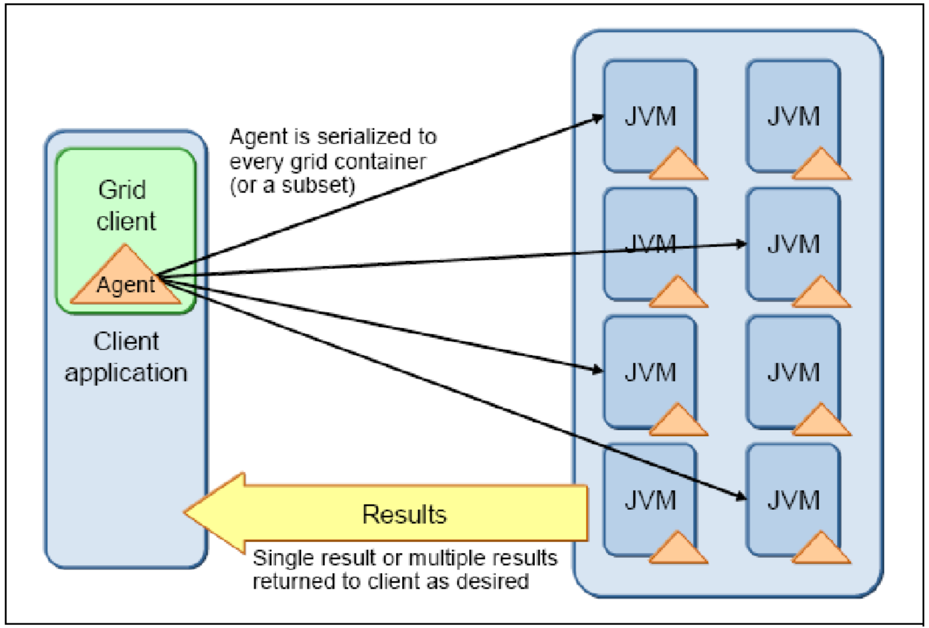
- Small, static table containing 3 rows are loaded into ALL partitions in the WXS grid
  - Queries that require ordering by the Whotype.precedence can be honored from ANY partition the query is executed
  - WhoType represents a type of user defined by the bank
  - WXS Agents are used to accomplish this goal

SELECT o FROM TOffering o join o.whotypeid as whot join o.wheretypeid as wheret WHERE o.feld = ?1 **ORDER BY whot.precedence ASC, wheret.precedence ASC**



Data Grid Agents

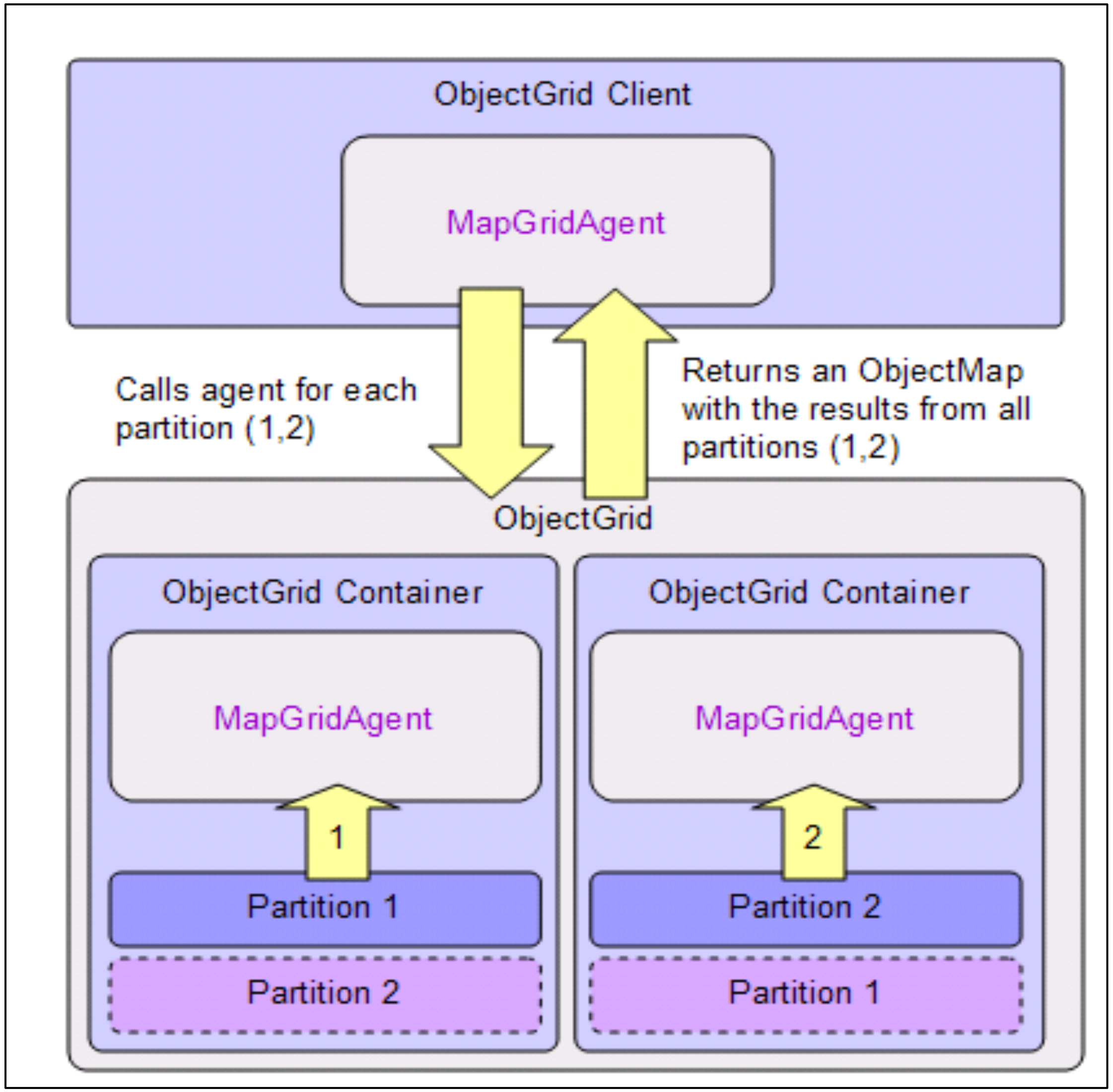
- Agents run in the WXS container process
- Perform some operation on cache entries in the container
- Returns result to the client
- Using DataGrid APIs clients can send **agents** to one, some or all partitions in a grid.
  - Client invocations may contain keys
    - WXS determines the set of partitions to which the agents will be routed based on the keys passed into the agent
    - If no keys are passed in, then agents will be serialized to all partitions



Agents

- The **agents run only on the primary shards** and each agent instance can see only the data located in that shard
  - The business logic is in the agent and data is local
    - No serialization/deserialization or network hop
- Extremely suitable for concurrent grid computations where:
  - The computation logic is identical in all the shards (logical parts)
  - There are no dependencies on the computations among each shards
  - Computations in each shard do not communicate with each other
- Can be used in some cases to process massive amounts in a partitioned grid utilizing the horse power of shard hosting machines.
- Throughput is dependent on the slowest executing primary partition for a computation.

Parallel Map Control Flow



Demonstration

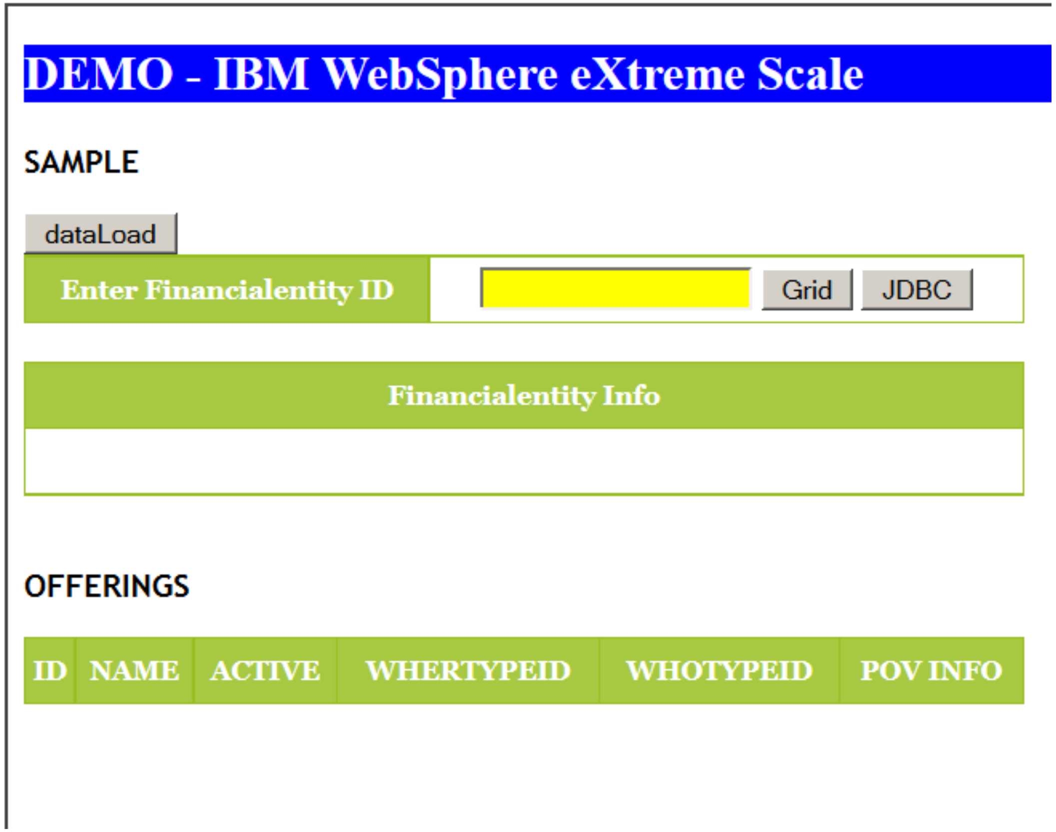
- **Pre-load the cache** with all of the data from the database
  - The following data partitioning techniques have been implemented to support single partition queries and proper ordering of the results:
    - The graph of data including the Financial Institution, Offering, and Offering Parameters are loaded into a single partition based on the hash algorithm of the Financial institution
    - The WhoType and Wheretype data is loaded into ALL partitions to allow fro the ORDER BY queries to be executed from any partition

- **In-line cache scenario** using custom JDBC loader plug-in
  - Custom JDBC loader plug-in will allow updates to an Offering to be synchronized to the backend database

- **WXS Cache Queries** used to return collections of Offerings and Offering Parameters to the caller
  - Efficient use of the WXS query API can be accomplished since the related graph data is co-located into the same partition as it's root object

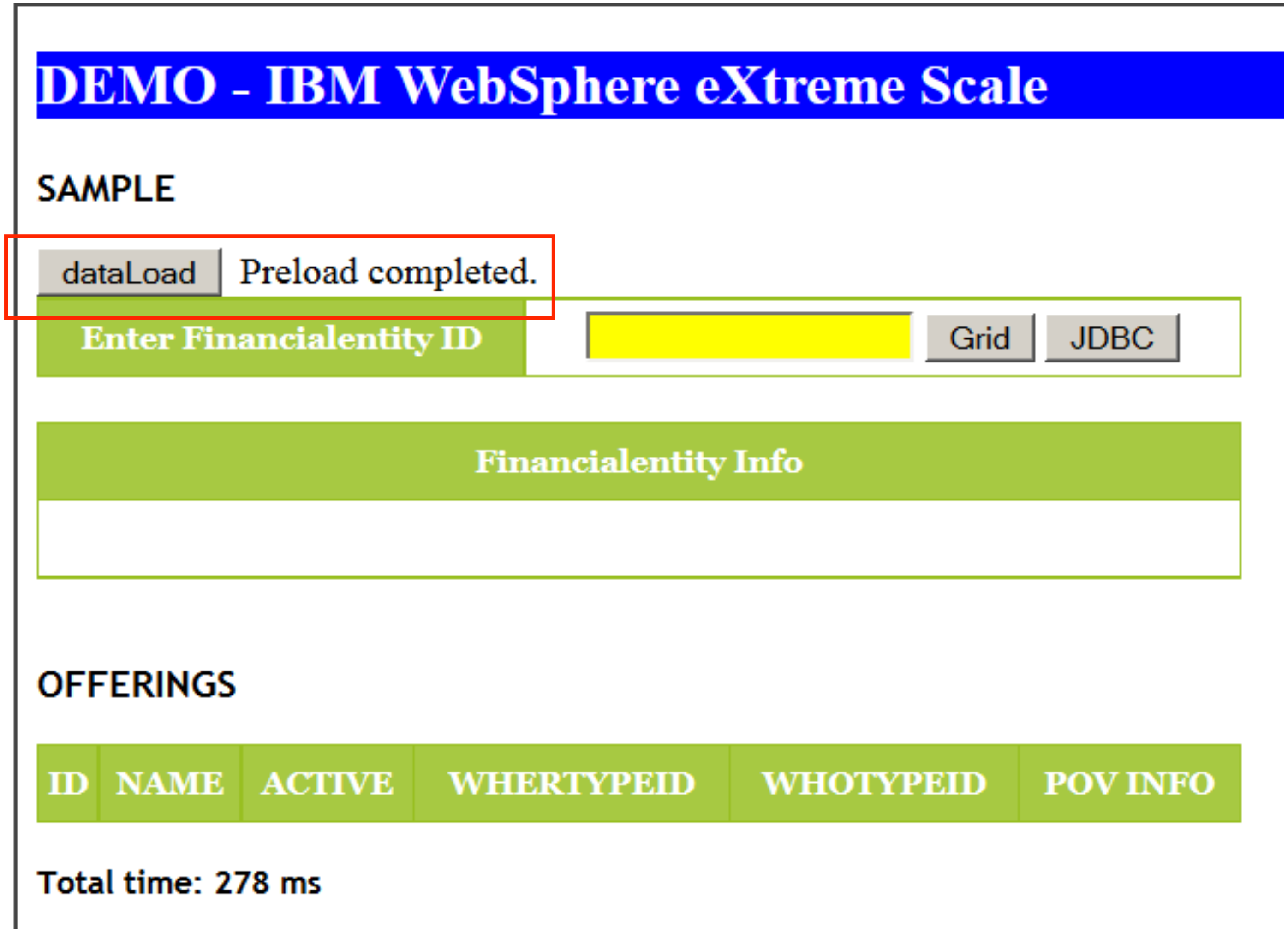
Demo: Sample JSP to illustrate the cache behavior

- Start with an empty cache



Demo: Sample JSP to illustrate the cache behavior

- Click the **dataLoad** button to preload the cache with contents from the database



Demo: Sample JSP to illustrate the cache behavior

- Get the Banks "Offerings" from Financial InstitutionID=33 from the cache

DEMO - IBM WebSphere eXtreme Scale

SAMPLE

dataLoad

Enter Financialentity ID	Grid JDBC

Financialentity Info
33

OFFERINGS

ID	NAME	ACTIVE	WHERTYPEID	WHOTYPEID	POV INFO
50	Balance - Primary Checking	1	2	2	Grid JDBC
49	Balance - Primary Savings	1	2	2	Grid JDBC
56	Balance - Savings #2	1	2	2	Grid JDBC
57	Balance - Savings #3	1	2	2	Grid JDBC

Demo: Sample JSP to illustrate the cache behavior

- Get the associated parameters (POVs) from “Offering ID=48” from the cache

Financialentity Info
33

OFFERINGS

ID	NAME	ACTIVE	WHERTYPEID	WHOTYPEID	POV INFO
48	Transfer Money	1	2	2	Grid JDBC

POVS

ID	OFFERING ID	VALUE
562	48	Minimum Transer amount: $10
563	48	Money Transfer must be in increments of $10
579	48	$10
586	48	$20
609	48	$40
608	48	$50
611	48	$100
610	48	Other Amount

Demo: Sample JSP to illustrate the cache behavior

- Update the cached data: Offering with ID=48
- Retrieve the Offerings again from the cache again to verify it was updated

DEMO - IBM WebSphere eXtre

SAMPLE

OFFERING

ID	NAME
48	Transfer Funds

update

Financialentity Info
33

OFFERINGS

ID	NAME	ACTIVE	WHERTYPEID	WHOTYPEID	POV INFO
50	Balance - Primary Checking	1	2	2	Grid JDBC
49	Balance - Primary Savings	1	2	2	Grid JDBC
56	Balance - Savings #2	1	2	2	Grid JDBC
57	Balance - Savings #3	1	2	2	Grid JDBC
58	Balance - Savings #4	1	2	2	Grid JDBC
48	Transfer Money	1	2	2	Grid JDBC
55	Check Available Credit	1	4	2	Grid JDBC

Demo: Sample JSP to illustrate the cache behavior

- Verify the database has been updated via the in-line cache Write-Behind loader