Querying and Aggregating Data in the Cache

Objectives

After completing this lesson, you should be able to:

- Develop Filters to subset cache data, including sorting and aggregation of data objects
- Construct Filters using ValueExtractors
- Develop custom aggregators, which can run in parallel in a Coherence cluster
- Extend objects to support data affinity
- Implement Java clients that can perform dynamic queries using the Coherence Query Language (CohQL)
- Improve filter and query performance using indexes, trace, and explain plans

Filters and Caches

- Caches are easily accessible via `get` operations, but *only* using well-known object keys and sets of keys
- Filters can access a cache using identities, object attributes, or combinations via the `QueryMap` interface
- `QueryMap` methods return a `Set` of objects based on a `Filter`

```
package com.tangosol.util;

public interface QueryMap extends Map {


    public abstract Set keySet(Filter filter);
    public abstract Set entrySet(Filter filter);
    public abstract Set entrySet(Filter filter, Comparator comparator);

}
```

QueryMap.java

7 - 3

In prior lessons, you examined how to create and access caches either returning all elements, or a specific element using that element's identity. However, often you need to access cache data based on other attributes of data, where you do not know an object's identity, or access data using both scenarios. The query map interface has two core functions:

- It provides the ability to perform simple queries against a cache. These queries are either for the keys of a set of objects, the entries themselves, of a sorted set of the elements.
- Coherence also provides indexing and parallelization features. Indixes are especially useful with partitioned caches. By adding servers to a partitioned cache, you can increase throughput (total queries per second) and reduce latency by reducing the time taken by queries.

In relational database terms, you can create indexes on a column. In Coherence, indexes are created by attribute. This improves the performance of the query just completed in a relational database.

Filter Execution

7 - 5

Examine the graphic in the slide and note the following:

- At the top left, observe that you have a client that executes a query.
- Coherence automatically executes this query in parallel on all the JVMs that store the data. In Coherence, you have multiple JVMs spread out across multiple physical servers. In this process, Coherence automatically spans a task on each of these JVMs, automatically does the filtering on the query, gets the results, and sends the result set back to the client.
- Note the word “Index” in the brown box in the graphic. Coherence has a simple built-in optimizer. If there is an index that you created, Coherence uses this index to get better performance from the query.
- The parallelization is done automatically, and you need not specify anything on the client. Coherence automatically optimizes and parallelizes the query. This means that in a typical Coherence deployment, the power of all the CPUs in your physical servers are being used to perform the query.

Developing Filters

Coherence provides a set of out of the box filters, but supports custom filters via the `Filter` interface:

```
package com.tangosol.util;
public interface Filter {
  public abstract boolean evaluate(Object obj);
}
```

The same filter we saw in events!

Filter.java

```
import com.tangosol.util.Filter;
public class InFlight implements Filter, Serializable {
    public boolean evaluate(Object o) {
        Flight flight= (Flight) o;
        return flight.isInFlight();
   }
}
```

InFlight.java

```
NamedCache flights = ...;
Set inflightFlights = flights.entrySet (new InFlight());
```

Application.java

7 - 6

Filters are nothing more than classes that implement the `com.tangosol.util.Filter` interface. This interface has a single method `evaluate` that returns true in the event the object meets the criteria of the filter.

In the example provided, the `InFlight` filter returns all `Flight` objects that are marked as such. An application then uses the filter against a cache to return objects that meet the filter criteria.

Accessing Object Properties

Filters can access object properties using method names.

```
public class Flight implements Entity<Long>, Comparable,. . . {
. . .
public boolean isInFlight() { return mbInFlight; }
. . .
}
```

Assuming this class...

... you can rewrite the prior example using a stock filter to access a method as

```
NamedCache flights = ...;
Set inflightFlights = flights.entrySet (
      new EqualsFilter("isInFlight", BOOLEAN.TRUE));
```

7 - 7

Filters can access object values via the getter methods as shown in the example in the slide.

In this example, the flights in progress are returned using an out-of-the-box filter known as the `EqualsFilter`. This filter takes a method name, in this case `isInFlight`, and compares its value to the provided one. Any objects that report the value provided for the method specified are added to the set.

Out-of-the-box Filters

Most expressions can be expressed in terms of an object's attributes and traditional logical operations such as AND, OR, NOT, equals, and others.

Filter	Description (com.tangosol.util.filter)
`EqualsFilter`	Compares a value extracted via a method to a value
`AllFilter`	Performs the logical *and* of a set of filters in a filter array
`ContainsFilter`	Returns `true` if the value extracted via a method contains the provided Object
`IsNullFilter`	Returns `true` if the value extracted via the method is null. See also `isNotNull`.
`LessFilter`	Returns `true` if the value exacted is less than the provided value. Returns `false` on null returns.

7 - 8

This slide shows a small subset of the supplied filters. The `com.tangosol.util.filter` package contains 34 filters, all of which can be combined using logical expressions to create complex cache filters.

A note on the Container filter: The associated value extractor must return a collection and the second argument is any Object. The collection is tested for containing that passed Object.

Example: Filtering with `EqualsFilter`

To filter a set of airports based on a specific country:

```
public class AirPort . . . {
. . .
    public boolean getCountry() { . . . }
. . .
}
```

Airport.java

```
. . .
import com.tangosol.util.filter.EqualsFilter;
. . .
NamedCache airports= ...;

Filter USAirports = new EqualsFilter("getCountry","US");
Set <Map.Entry> usAirports= airports.entrySet (USAirports );

for (Map.Entry entry: entries) { . . . }
```

Application.java

7 - 9

Using the out of the box `com.tangosol.util.filter.EqualsFilter` to select a specific set of airports:

1. Create a filter using the `getCountry` method exposed by the `AirPort` class. This filter uses the `String` constructor of the `EqualsFilter` object to select “US” airports.
2. Using the filter, obtain an entry set, note that entry sets return `Map.Entry` elements, and this example qualifies the result.
3. Process the results as required.

Note that the `EqualsFilter` provides constructors for a variety of types other than `String`. See the JavaDoc for a complete set.

Sorting

Result sorting on Filters happens in two ways:

- **Natural:** Using the object's `Comparable` implementation
- **Specific:** Using a provided `Comparator`

```
package com.tangosol.util;

public interface QueryMap extends Map {
    . . .
    public abstract Set entrySet(Filter filter,
                                 Comparator comparator);
    . . .
}
```

QueryMap.java

7 - 10

Objects returned by Filter operations are sorted in one of three specific ways:

- Random, and not `Comparable`: If an object does not implement the `Comparable` interface, the ordering of returns based on filter is often the order that the objects were entered into the cache based on the hash of the key. This is most likely a result of differing cache topologies. Objects can come from more than one node, in which case the order is effectively random.
- Order based on `Comparable`: If an object does implement the `Comparable` interface, and no overriding `Comparator` is provided, then the objects will be returned in what is considered a *natural* order. Natural order is the order the objects enforce on themselves.
- Specific order: If a `Comparator` is provided, regardless of whether the object implements `Comparable`, then objects are returned based on the order defined by the `Comparator` itself.

Example: Sorting with a `Comparator`

```
. . .
public class AirportComparator implements java.util.Comparator {
    public int compare(Object first, Object second) {
      AirPort firstAP = (AirPort) first;
      AirPort secondAP = (AirPort) second;
      if (firstAP.getCountry().equals(secondAP.getCountry())) {
        return firstAP.getCity().compareTo(secondAP.getCity());
      } else {
        return firstAP.getCountry().compareTo(secondAP.getCountry());
      }
   }
}
```

Orders by city within country

airportcomparator.java

```
. . .
NamedCache airports= ...;
. . .
Filter USAirports = new EqualsFilter("getCountry","US");

Comparator apc = new AirportComparator();
Set <Map.Entry> usAirports= airports.entrySet (USAirports,apc);
. . .
```

Application.java

7 - 11

Writing a Comparator is a simple process if you understand the return results. Comparators are defined to provide a complete ordering of elements. The return is one of the values based on the provided objects. It returns:

- **A negative value** if the first object is less than the second. The exact value is unimportant, just that it is less than zero.
- **Zero** if the first object is equal to the second object.
- **A positive integer** if the first object is greater than the second object. The exact value is unimportant, just that it is more than zero.

Paging and `LimitFilter`s

`LimitFilter`s

- Differ from other filters that are created, used, and discarded
- Take a traditional filter and a number of results to return per *page* as arguments
- Use `nextPage()` to return the next N results

```
. . .
NamedCache airports= ...;
Filter USAirports= new LessEqualsFilter("getCountry","US");
Filter pagedAirports  = new LimitFilter (USAirports, 5);
. . .
Set <Map.Entry> usAirports = null;
do {
    usAirports= airports.entrySet (pagedAirports,. . .);
    if ( usAirports.isEmpty()) break;
    . . .
    pagedAirPorts.nextPage();
} while (true);
```

Application.java

7 - 12

One filter that requires special notice is the `LimitFilter`. A `LimitFilter` is not really a filter at all, but rather acts as a buffer returning sets of objects based on a set limit. When creating a limit filter, another filter is provided, along with a count of objects to return. Then, executed by a `entrySet` operation, the filter defines what elements to return, and the limit filter defines how many to return. Second and subsequent sets of values are returned via the `getPage`() method, and the re-executing the `entrySet` operation.

Aggregating Results

- Aggregation is the process of applying a process or function to a set of values to retrieve a single result, such as a sum or average.
- Filters can be used to aggregate by iterating over all values, but force the return of all data to a single process which may result in:
  - High network traffic due
  - Single threaded bottle neck in the processing application
- Aggregators allow the process to be executed on each node.

```
public abstract class AbstractAggregator
           implements InvocableMap.ParallelAwareAggregator {
. . .
}
```

AbstractAggregator.java

7 - 13

Aggregating results is a common operation in most database applications. Typical aggregations include sums, averages, minimums, maximums, oldest, and so on. Performing aggregations with Filters is a simple process, but loses some of the distributed benefits of Coherence by pushing all data to a single process. `EntryAggregator`s solve this problem by allowing developers to push processing to the nodes containing the data and then return a final result. With aggregators, each node calculates a partial result, and then aggregates the partial result into a single final result.

Understanding Aggregation Execution

7 - 14

In theory, aggregators can run on any node in the cluster. Each cluster node then processes a portion of the aggregation, and returns its values back for final aggregation.

The aggregation process is where each cluster node processes its portion of the aggregation. For example, if a set of flights was being distributed across all members of the cluster, each cluster member would have some portion of the total flights. The local node would then aggregate its portion of flights and send its partial result back to the requestor. The original requestor then aggregates all the partial results.

Each time a call is made to process a result, it can be in one of two modes: final or intermediate. When an intermediate call is made, the aggregation code simply aggregates the next value provided to it. If the call is final, the aggregator is actually processing individual aggregation rollup results.

Developing Aggregators

Aggregators are typically developed against `AbstractAggregator`, which supports parallel processing and provides three methods that must be implemented.

```
public abstract class AbstractAggregator
          implements InvocableMap.ParallelAwareAggregator {

    public AbstractAggregator(ValueExtractor valueExtractor){ }
```

Should call super(valueExtractor) to store the provided value extractor.

```
    protected abstract void init(boolean isFinal);
```

Initializes the aggregation class. Here is where context variables are initialized.

```
    protected abstract void process(Object value, boolean isFinal);
```

Called to perform processing

```
    protected abstract Object finalizeResult(boolean isFinal);
```

Called to return the final result

```
 }
```

AbstractAggregator.java

7 - 15

Technically, aggregators are developed by extending the com.tangosol.util.InvokableMap.EntryAggregator interface. However, in practice, they implement the `AbstractAggregator` class, which provides a set of common code that can be used to streamline the development processing.

Aggregators developed this way must implement three methods and a constructor:

- **Constructor:** Each aggregator must implement a constructor that `AbstractAggregator(ValueExtractor valueExtractor)` mimics by calling super.
- `init`: Initializes and temporary variables
- `process`: Performs any required processing, such as keeping running totals, and so on
- `finalResult`: Returns the end result of the processing

Main Versus Parallel Execution

Aggregators run as either main or parallel.

- Main aggregators: Responsible for summarizing partial results
- Parallel aggregators: Responsible for creating partial results

7 - 16

Aggregators run in two specific ways: main and parallel. Understanding the distinction is crucial to correct implementation.

In main mode, the aggregator is started via an application. The main aggregator may aggregate results, acting in parallel, if on a storage-enabled node with content, or as the diagram shows, acts to roll up partial results to provide the final aggregation.

In parallel mode, the aggregator takes a set of values, based on a Filter, and a value extractor, and creates intermediate results.

Because the same class is used in both scenarios, a mechanism must be provided to determine which mode to execute. The mechanism for this is a simple flag passed to each of the three aggregator methods, `isFinal`, which tells the method which set of semantics to execute.

Working with `isFinal`

Method	Semantics	Object Argument/return
`init`	Has a single purpose, initialize the aggregator	NA
`process (false)`	The method should execute to construct a partial value, such as adding to a total	Object contents of type returned by value extractor passed by constructor
`process (true)`	The method should perform a roll up of partial values	Object is of the value internally defined by the aggregator
`finalize Result (false)`	The method should return the final result of a partial aggregation	Returns a roll up partial result. The type is often an internal use only private class.
`finalize Result (true)`	Return the result of an entire aggregation	Returns a final result, the type being appropriate for the aggregation, such as a Double and so on

7 - 17

Understanding how to interpret the `isFinal` flag is crucial to the development of aggregators.

The `isFinal` flag dictates how each method should function.

- `init`: The state, except in some specialized cases, is ignored and the aggregator should simply initialize to an appropriate value of zero.
- `process (false)`: This is the normal case. The aggregator should simply total results as required. In this case, the object passed to the aggregator is defined by the `ValueExtractor` passed to the constructor.
- `process(true)`: In this case, there are no more results to roll up, and the aggregator is being used to perform rollup processing. The value passed is an intermediate rollup result, which can be anything as defined by the aggregator.
- `finalizeResult(false)`: Returns a partial result as dictated by the aggregator
- `finalResult(true)`: Returns the end result of aggregating a set of partial results

Example Aggregator

```
import com.tangosol.util.ValueExtractor;
import com.tangosol.util.aggregator.AbstractAggregator;

public class StringAggregator extends AbstractAggregator {

    transient StringBuffer results;

    public StringAggregator() { super(); }

    public StringAggregator(ValueExtractor extractor) {
      super(extractor);
    }

    protected void init(boolean isFinal) {
        results = new StringBuffer();

    }

. . .
```

StringAggregator.java

7 - 18

This slide shows the constructor and initialization methods of a string aggregator. The string aggregator takes a string field from an entity, which must be returned by the value extractor passed to the constructor, and compiles a list of values.

Example Aggregator

```
    protected void process(Object object, boolean isFinal) {
      if (object == null ) return;
      if (!isFinal) {
          if (results.length() == 0) {
              results.append((String)object);
           } else {
              results.append(",");
              results.append((String)object);
           }
       } else { // isFinal
            // basically the same thing
           // but with an entire set of values
           if (results.length()>0) {
               results.append(",");
            }
            results.append((String)object);
       }
   }
   protected Object finalizeResult(boolean isFinal) {
       return results.length() > 0? results.toString():null;
    }
}
```

StringAggregator.java

7 - 19

The process method runs in either main or parallel mode as indicated by the `isFinal` method. Because of the simplicity of this aggregator, the behavior is essentially the same. Finally, when the parallel aggregations are complete, the result if any, is returned.

Using Aggregators

Aggregators are started against a cache using one of the `InvocableMap.aggregate()` methods that `NamedMap` extends.

```
package com.tangosol.util;
public interface InvocableMap extends Map {
. . .
    public abstract Object aggregate(Collection collection,
             EntryAggregator entryaggregator);
    public abstract Object aggregate(Filter filter,
        EntryAggregator entryaggregator);
. . .
```

```
NamedCache airports= ...;

Filter le = new EqualsFilter("getCountry","US");

Object result = airports.aggregate(le,
      new StringAggregator(new ReflectionExtractor("getCity")));
```

7 - 20

Aggregators are used by executing one of the aggregate methods against the `InvocableMap` interface. Named cache, as well as other caches, extends this interface so the methods are available for use.

The two forms are similar, one executing against an existing collection, and one against a filter. Each takes a class that extends the aggregator interface, and requires a value extractor.

In the example shown, the reflection extractor is used to access the city field of the AirPort object. Then, aggregator returns a string which is a collection of all the cities names based on the provided filter, which selects only US airports.

Out-Of-the-Box Aggregators

A selection of out-of-the-box aggregators:

Aggregator	Description (`com.tangosol.util.aggregator`)
`DoubleSum`	Sums up numeric values extracted from a set of entries in a Map
`Count`	Calculates the number of entries
`LongMin`	Calculates the minimum of a set of entries
`LongMax`	Calculates the maximum of a set of entries
`LongSum`	Calculates the sum of a set of entries
`DistinctValues`	Returns the set of unique values extracted from a set of entries in a Map

7 - 21

This slide is a partial list of provided aggregators. Students are directed to the documentation of the `com.tangosol.util.aggregator` package for a complete list.

Filters and Indexes

- Each application may suggest the same set of indexes when they start.
- There is no downside to suggesting an index regardless of whether another application has already suggested the same index.
- `addIndex()` is a no-op for existing indexes.
- Indexes are maintained by Cache Entry Owners. That is, for partitioned topology, the primary partitions maintain their own indexes.
- A sorted index affects the performance of filters.

7 - 22

As discussed earlier, you can search for cache entries that meet specific criteria. You can index any queryable attribute with the `addIndex()` method. This method serves only as a hint to the cache implementation. This can be ignored by the cache if the indexes are not supported or if the desired index already exists.

Each application using Coherence may suggest the same set of indexes when it starts. However, there is no downslde to executing an `addIndex()` method because it is a “no operation” (no-op) if the index already exists.

For example, in a partitioned topology, the primary partitions maintain their own indexes. Coherence knows when indexes are “on” for the attributes, and will act accordingly. Indexing requires the ability to extract attributes on each partitioned cache node. For dedicated CacheServer instances, this usually implies that the application classes must be installed in the CacheServer `CLASSPATH` environment variable.

For local and replicated caches, queries are evaluated locally against unindexed data. For partitioned caches, queries are performed in parallel across the cluster, using indexes if available. Coherence has a built-in Cost-Based Optimizer (CBO) to prioritize the usage of indexes. A sorted index determines how well filters perform, and can potentially speed up

Index: Example with Ordering

- Suggest an index for trades based on their stock symbol.
- Queries that use this index are sorted (such as an `ORDER BY`) and use natural ordering (therefore the `null`).

```
NamedCache trades = . . .

trades.addIndex(
    new ReflectionExtractor("getSymbol"),
                              true, /*sort */
                              null);
Filter allA = new LikeFilter("getSymbol", "A%");

Set stocks= trades.entrySet(allA);
```

7 - 24

The slide shows an example of adding an index. To add an index:

1. Obtain a handle on `NamedCache`.
2. Before you execute a query, you need to add an index.
3. Create a new `ReflectionExtractor` on the `getSymbol`() method. The `addIndex`() method indexes the result set of this method. This is similar to indexing the symbol column in relational databases. Pass a Boolean value of `true` to sort the result set.

This example does not include an optional comparator. If you do not pass a comparator, Coherence sorts based on what Java thinks is the natural order. For a numeric column, it sorts in the ascending order, that is, smallest to highest number. Depending on the language that you use with the string, it sorts based on the natural ordering of that particular string. So, it is in alphabetical order in this case.

Now that you have created the index, you execute a query that includes the `entrySet` method. In this case, because you are using `LikeFilter`, the method will query all symbols that begin with the letter “A.”

Index: Example without Ordering

- Suggest an index for trades based on their market.
- No specified order

```
trades.addIndex(
      new ReflectionExtractor("getMarket"),
                false,   /* do not sort */
                null);
```

7 - 26

When Coherence gives the result set back to the client, it is sorted by alphabetical order.

If you use the `addIndex()` method in your application, and the index already exists, it is essentially a hint to the optimizer. By calling `addindex`, you tell Coherence to use the index if it exists. If it does not exist, you instruct Coherence to create and use the index.

In this example, you perform the following:

- You add an index on the `getmarket()` method.
- Because you do not plan to sort the result set, you set the Boolean parameter to `false`.
- You set the optional comparator parameter to `null`.

What is CohQL?

Coherence Query Language, or CohQL:

- Is a lightweight syntax, in the tradition of SQL
- Is used to perform operations on a Coherence cache
- Can be used either programmatically or from the command-line
- Includes:
  - `QueryHelper` API: Supporting programmatic construction of filters using a SQL-like syntax
  - `QueryPlus`: A command-line tool for cache access

```
SELECT * FROM "orders" WHERE 40 > price
```

7 - 27

Introduced in Coherence 3.6, Coherence Query Language or CohQL defines statements to manage data against a cache. CohQL enables you to write clear, concise query statements, that are translated to Filter instances and executed against a cache.

CohQL defines a data model based on the cache objects and their embedded objects, which can be traversed using dot notation (.). CohQL can traverse the schema given to the entities and their fields, both of which are mapped to an underlying cache.

CohQL defines operators and expressions based on the data model using a SQL-like syntax.

CohQL Statements

A CohQL statement can be:

- A `SELECT` statement, which might include various clauses, such as `FROM`, `WHERE`, and `GROUP BY`
- An `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `BACKUP`, or `RESTORE` statement

No order by, but can use comparators!

BNF Grammar (Backus-Naur Form):

```
select_clause ::= SELECT select_expression {, select_expression}*
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
create_clause ::= [CREATE | ENSURE] CACHE "cache_name"
```

Examples:

```
SELECT * FROM Publisher WHERE name ="Macmillan"

UPDATE "employees" SET salary = 1000, vacation = 200 WHERE grade > 7

DELETE FROM "employee" WHERE bar.writer IS NOT "David"

CREATE CACHE "employee"
```

7 - 28

CohQL query language supports the same four types of statements most SQL dialects support: INSERT, SELECT, DELETE, and UPDATE. Additionally, CohQL supports the ability to create caches directly in CohQL as well as backup, restore, and otherwise manage caches.

As the slide shows, basic CohQL statements are very similar to their SQL counterparts in both form and function.

A noted difference is the absence of a `JOIN` clause. CoQL is *not* SQL and does not support joining two caches together. Additionally CohQL does not support `ORDER BY`.

Working with Queries

The `QueryHelper` factory class provides five methods for creating queries and one for creating a value extractor.

```
package com.tangosol.util;
public class QueryHelper {
   public static Filter createFilter(String s);

   public static Filter createFilter(String s, Object aBindings[]);


   public static Filter createFilter(String s, Map mapBindings);


   public static Filter createFilter(String s, Object aBindings[],
                                               Map mapBindings);


   public static ValueExtractor createExtractor(String s);
}
```

7 - 29

The Query helper factory class provides a set of five methods, four of which are dedicated to creating Filter representations of the provided CohQL statement.

Of particular note are the binding variables. Coherence supports both numeric and named parameter bindings in CohQL statements. Variables of type `Object[]` are arrays of bindings using zero-based variable replacements of the form `?#`. Variables of type `Map` are named variable replacements of the form `:`*`name`*, each of which will be discussed in more detail in later slides. An important point is that `?#` variables are 1-based, but `Object[]` is zero-based, as such element 0 maps to ?1.

Note that each of these methods can throw a `FilterBuildingException`.

General SELECT Syntax

- Select statements are composed of:

```
SELECT (properties* aggregators* | * | alias)
FROM "cache-name" [[AS] alias]
[WHERE conditional-expression] [GROUP [BY] properties+]
```

- The only required portion of a select is SELECT…FROM…

```
SELECT order FROM "orders"
```

When using QueryHelper, '*SELECT * FROM "cachename" WHERE*' is implied.

7 - 30

Select statements follow a simple core syntax. At a minimum, there is `SELECT` followed by a `FROM <entity> AS <alias>`. Only the select and from portions are required.

Additionally, a `WHERE` clause can be used to refine results, and `GROUP BY` can be used to group sets of returned records, often to perform operations on the set, such as averages and so on.

The asterisk (*) character represents the full object instead of subparts. It is not required to prefix a path with the cache-name. The `FROM` part of the `SELECT` statement targets the cache that forms the domain over which the query should draw its results. The *`cache-name`* is the name of an existing cache.

Path Expressions

Relations between objects can be traversed using a Java-like syntax known as a *path expression*.

Path expressions:

- Reference entities using alias or identification variables
- Reference properties using dot (`.`) followed by property name
- Can reference properties in an embedded field directly
- Are backed by `ReflectionExtractor`s that are prepended with `get` and upper casing the first letter of the identifier

```
SELECT order
    FROM Orders AS order
        WHERE order.shipped = FALSE and order.address.city="Boston"
```

7 - 31

Path expressions are a simple dot-separated (.) notation for accessing properties of entities. In its simplest form, a property is referenced by its owner alias, a dot (.), and the property name or accessor name without the "`get`". For example, in the slide, the `Order` entity has a property `shipped`. The property can be used in a `WHERE` clause via its alias and name. For example, in Java, you might have:

```
Order order = …
 if ( order.getShipped() == FALSE) { // do something };
```

which in a `WHERE` clause translates into:

```
 WHERE order.shipped = FALSE
```

Additionally, path expressions can nest. For example, assuming `Orders` have `getAddress`() and `Addresses` have `getPostalCode()`, you might write:

```
WHERE order.address.postalCode = '01805'
```

Note: The path expressions that cannot translate are style JavaBean methods. However, you can replace an access name with a method, such as `isSupported()`, which will be used directly and without conversion.

Filtering with WHERE

A WHERE clause typically has some number of conditions that can subset a SELECT.

- Value comparison using operators

<, <=, =, >, >=, <>	Boolean values and entity expression use only = and <>.

- Like comparison

LIKE "*string*"	Supports string wildcards _ and %

- Null comparison

IS [NOT] NULL	Is the variable [not] null?

- Contains

path-expression CONTAINS "literal"	Does the expression contain the literal?

- In comparison

path-expression IN (<*list of values*>)	Is value in set?

7 - 32

Operator precedence within the WHERE clause is as follows:

1. Path operator (.)
2. Unary + and -
3. Multiplication (*) and division (/)
4. Addition (+) and subtraction (-)
5. Comparison operators: =, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, CONTAINS [ALL | ANY]
6. Logical operators (AND, OR, NOT)

The WHERE clause supports only arithmetic at the language level.

The BETWEEN operator can be used in conditional expressions to determine whether the result of an expression falls within an inclusive range of values. Numeric, or string expressions can be evaluated in this way. The form is: BETWEEN *lower* AND *upper*.

Note that CohQL does not support the ORDER BY clause of traditional SQL.

QueryFilter: Example

- Select all employees who live in Massachusetts, but work in another state:

```
String where = "homeAddress.state = 'MA' and " +
               "workAddress.state != 'MA'";
Filter filter = QueryHelper.createFilter(where);
```

- Which is equivalent to:

```
Filter  filter =
AndFilter(
    EqualsFilter(
      new ChainedExtractor("getHomeAddress.getState", "MA"),
    NotEqualsFilter(
      new ChainedExtractor("getWorkAddress.getState"),"  MA")));
```

WARNING: Use queries sparingly! They do not scale well, and can cause considerable network traffic!

7 - 34

Caution: Queries can cause considerable network traffic across the entire cluster. It is suggested that developers first use a partition filter to reduce load, and then perform a query.

`key()` and `value()` Pseudo functions

- Developers may store data within the key of an object rather than duplicate the data inside the object itself.
- The `key()` pseudofunction can be used to reference an object's key and extract data from it. For example:

```
SELECT key().firstName, key().lastName
    FROM "ContactInfoCache"
      WHERE homeAddress.state IS 'MA' AND
            workAddress.state != "MA"
```

- The `value()` pseudofunction represents the complete contents of a object and can be used as:

```
SELECT key(), value() FROM employees
```

7 - 35

The `key()` and `value()` pseudofunctions reflect the fact that CohQL is not SQL and that Coherence is not a Database. These functions exist because you are dealing with caches that reflect the Map interface. Because you are dealing with caches, you need a way to handle the abstractions of keys and their stored values in a concrete way. The `value()` pseudofunction is in fact an `IdentityExtractor`, and the `key()` function is actually a wrapper around `KeyExtractor`.

Aggregating in SELECT

- Supports aggregation via a variant of the SELECT syntax
- CohQL supports aggregation functions:
  - COUNT
  - AVG
  - MIN
  - MAX
  - SUM
- Example:

```
SELECT supplier,SUM(amount),AVG(price)
    FROM "orders"
       GROUP BY supplier
```

7 - 36

An aggregate query is a variation on the SELECT query. Use an aggregate query when you want to group results and apply aggregate functions to obtain summary information about the results. A query is considered an aggregate query if it uses an aggregate function or possesses a GROUP BY clause. The most typical form of an aggregate query involves the use of one or more grouping expressions followed by aggregate functions in the SELECT clause, paired with the same lead grouping expressions in a GROUP BY clause.

Parameters

Parameters are identified in queries:

- By name: ':' + *name*, for example, ':po_Id'
- By position: '?#“, for example, '?1'

```
QueryHelper.createFilter("number = ?1", new Object[]{new Integer(42)};

HashMap env = new HashMap();
env.put("iNum",new Integer(42));

QueryHelper.createFilter("number = :iNum", env);
```

Replaced with

Replaced with

7 - 37

Parameters are defined in one of two ways:

- **By position:** Positional parameters are specified using parameter names using the `?1` syntax. Each parameter is prefixed with `'?'` and an increasing integer value, and then uses the parameter in the associated CohQL query. Remember, positional variables are 1-based, but the `Object[]` is zero-based.
- **By name:** Parameters can be specified by name as well as by using the colon (`:`) followed by a case-sensitive name. The `Map` variants of `createFilter` are then used to specify a value at run time.

Managing Cache Life Cycle

Caches can be created, deleted, backed up, and restored via CohQL.

- To create a cache

```
CREATE CACHE "cache-name"
```

- To use an existing cache in a CohQL script

```
ENSURE CACHE "cache-name"
```

- To backup a cache

```
BACKUP CACHE "cache-name" [TO] [FILE] "file-name"
```

- To restore a cache

```
RESTORE CACHE "cache-name" [ FROM ] [ FILE ] "filename"
```

- To delete a cache

```
DROP CACHE  "cache-name"
```

7 - 39

Note that these methods are ONLY available when running against a development Coherence cluster. The Coherence query console will only attach to a development cluster.

UPDATE Syntax

- UPDATE statements are composed of:

```
UPDATE "cache-name" [[AS] alias]
SET update-statement {, update-statement}*
[ WHERE conditional-expression ]
```

- The only required portion of a select is UPDATE ...FROM...

```
UPDATE "employees"
    SET salary = 1000, vacation = 200
        WHERE grade > 7
```

7 - 40

Each update statement consists of a path expression, assignment operator (=), and an expression. The expression choices for the assignment statement are restricted. The right side of the assignment must resolve to a literal, a bind-variable, a static method, or a new Java-constructor with only literals or bind-variables. The UPDATE statement also supports the use of aliases.

INSERT Syntax

- CohQL fully supports INSERT into a cache, based on a key.
- Insert keys must either be:
  - Provided via the INSERT statement, or
  - Generated object must support getKey()
- Insert statements are composed of:

```
INSERT INTO "cache-name"
[ KEY (literal | new java-constructor | static method) ]
VALUE (literal | new java-constructor | static method)
```

- Example:

```
insert into "employee" key "writer" value "David"
```

7 - 41

The insert command can be used to insert elements into a cache, even if those elements are java class based. In the example show strings are shown being inserted into a cache however either the key or value can be a java constructor statement such as `new Customer("x","y")` or a static factor method which returns an object.

DELETE Syntax

- CohQL supports cache object removal via DELETE.
- Delete syntax:

```
DELETE FROM "cache-name" [[AS] alias]
[WHERE conditional-expression]
```

- Delete fully supports SELECT … WHERE clause syntax and aliases.

7 - 42

The delete statement can be used to remove cache elements directly. Any valid where clause can be used to remove an element or set of elements.

Index Management

CohQL supports the creation and removal of indexes.

- To create or ensure the use of an index:

```
[ CREATE | ENSURE ] INDEX
    [ON] "cache-name" (value-extractor-list)
```

- To drop an existing index:

```
DROP INDEX [ON] "cache-name" (value-extractor-list)
```

- The value extractor list:
  - Is a comma-separated list
  - Uses path expressions to create `ValueExtractor`s
  - Creates a `MultiExtractor`, if a list of path expressions is specified

7 - 43

The index command is used to create or ensure an index is available. Index is particularly useful when coupled with TRACE or EXPLAIN PLAN. To use index with TRACE or EXPLAIN PLAN simply examine a queries performance using either, add an index, and re-examine the performance to see how it may have changed.

Starting `QueryPlus`

To start an instance of Oracle Coherence CohQL tool:

1. Open a command prompt to `%COHERENCE_HOME%/bin`.
2. Ensure that `java_home` is set appropriately.
3. Run `query.[cmd|sh]`.

```
C:\oracle\coherence\bin>query.cmd
** Starting storage disabled console **
java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02)
Java HotSpot(TM) Server VM (build 16.3-b01, mixed mode)

Coherence Command Line Tool
jline library cannot be loaded, so you cannot use the arrow
keys for line editing and history.

CohQL>
```

Warning: QueryPlus is a development tool and not intended for production use!

7 - 44

The CohQL command-line tool provides a non-programmatic way to interact with caches by allowing statements to be issued from the command line. The tool can be run using the `com.tangosol.coherence.dslquery.QueryPlus` class or, for convenience, a startup script is available to run the tool and is located in the `COHERENCE_HOME/bin/` directory. The script is available for both Windows (`query.cmd`) and UNIX (`query.sh`).

`QueryPlus` Arguments

`QueryPlus` supports the following arguments:

Argument	Description
`-t`	Enables trace mode debugging output
`-c`	Exits after executing the command-line arguments
`-s`	Runs in silent mode. Often used with I/O redirection
`-e`	Runs the command-line tool in extended language mode. This mode allows object literals in update and insert commands.
`-l` *`statement`*	Executes the given statement. Statements must be enclosed in single or double quotes. Any number of `-l` arguments can be used together.
`-f` *`filename`*	Processes the statements in the given file. Statements must be separated by semicolons (;). Any number of `-f` arguments can be used.

7 - 45

Query Optimization: Explain Plans and Trace

- Query performance is directly related to cache search order, and index use during query execution.
- Queries can be examined by using `EXPLAIN PLAN` and `TRACE` to determine cost and effectiveness:
  - `EXPLAIN PLAN` – An estimated cost of the query
  - `TRACE` – The actual cost of query execution
- Queries can be evaluated at the command line or in code.

Form

```
EXPLAIN PLAN FOR select statement | update statement | delete statement
TRACE select statement | update statement | delete statement
```

Example:

```
TRACE SELECT * from "MyCache" WHERE age=19
```

7 - 46

Before Coherence 3.7.1, there was no way to determine the performance of a query other than writing timing code, or timing a query by hand. Neither of these was optimal or even convenient. In addition, it was impossible to determine which portions of the query had the most impact. For example, did a specific `and` filter have more impact on the query performance than a `like` filter? Would an index on one of the fields change the performance?

Coherence 3.7.1 introduced two new features to support static and dynamic evaluation of query performance. These two features are:

- Explain plans – A static analysis of a query, producing an estimated cost for the query
- Trace – A dynamic analysis of the query, producing an actual effectiveness rating for a query

Using explain plans and trace statements, developers can examine how queries are executed, in what order clauses are executed, and the effect of a given index. Both of these features can be executed from either the command line or from within an applications code, providing an easy-to-use robust mechanism for examining the performance of a given query.

Determining Query Performance

- Query performance:
  - Type is defined by `RecordTypes` ← EXPLAIN or TRACE
  - Is examined using `cache.aggregate(...)`
  - Is executed via a `QueryRecorder`
  - Is returned via `QueryRecords`

```
import com.tangosol.util.QueryRecord;
import com.tangosol.util.aggregator.QueryRecorder;
import static com.tangosol.util.aggregator.QueryRecorder.RecordType;
import com.tangosol.util.QueryHelper;

. . .
NamedCache cache = . . . ;
String whereClause = "some query…";
Filter filter = QueryHelper.createFilter(whereClause);

QueryRecorder agent = new QueryRecorder(RecordType.TRACE);
QueryRecord record = (QueryRecord) cache.aggregate(filter, agent);


System.out.println(record);
```

7 - 47

Applications can examine query performance statically or dynamically by using `RecordTypes`. The general code to examine a query's performance is:

1. Create a filter. In the example shown, the `QueryHelper` class turns the actionable portion of a `where` clause into a `Filter`.
2. The `QueryRecorder` class is a type of aggregator, which is capable of querying all nodes in the cluster and aggregating the results. The recorder takes one argument, the `RecordType` of which is either `EXPLAIN` or `TRACE`, depending on need.
3. The filter and the query recorded are then executed against a cache using `cache.aggregate()`, which returns a generic `Object`. In the case of a `QueryRecorder`, the result is a `QueryRecord`.
4. The code prints the result of the operation.

Understanding EXPLAIN PLAN Output

Explain Plan output is broken up into two parts:

- Explain Plan – A breakdown of a query by filter component showing associated index and cost

```
Explain Plan
Name                                        Index          Cost
=====================================================================
com.tangosol.util.filter.AndFilter         | ----          | 0
  LikeFilter(.getFirstName(), Al%)         | 0          | 2300000
  com.tangosol.util.filter.AndFilter       | ----          | 0...
```

Indents represent nesting in query.

- Index – List of indexes associated with an explain plan, including the extractor used to access the data

```
Index Lookups
Index Description      Extractor          Ordered
=================================================================== 0
        No index found .getFirstName()    false
```

7 - 48

Explain plan output describes, for each connector, and/or/not, and each specific data-based filter, the index and cost of the filter. Rows associated with indexes are inherently less costly because they allow for more efficient data selection. However, an index may have more or less impact due to location within a filter. For example, an index applied late in the execution, or because the data falls outside the range of the filter element (ages < 16 or > 50, for example) might have little impact because it affects a small number of result rows.

Columns are:

- **Name** – This column shows the name of each filter in the query. Composite filters show information for each of the filters within the composite filter.
- **Index** – This column shows whether or not an index can be used with the given filter. If an index is found, the number shown corresponds to the index number on the Index Lookups table.
- **Cost** – This column shows an estimated cost of applying the filter. If an index can be used, the cost is given as 1. The value of 1 is used because the operation of applying the index requires just a single access to the index content. If no index exists, the cost is calculated as `EVAL_COST *` number of keys. The `EVAL_COST` value is a constant value and is 1000. This is intended to show the relative cost of doing a full scan to reduce the key set using the filter.

EXPLAIN PLAN Output: Example

Example of Explain Plan output for query
"firstName like 'Al%' and (age > 16 and age < 50)"

```
QueryRecorder agent = new QueryRecorder(RecordType.EXPLAIN);

Explain Plan
Name                                        Index          Cost
=====================================================================
com.tangosol.util.filter.AndFilter         | ----          | 0
  LikeFilter(.getFirstName(), Al%)         | 0          | 2300000
  com.tangosol.util.filter.AndFilter       | ----          | 0
    GreaterFilter(.getAge(), 16)           | 0          | 2300000
. . .
Index Lookups
Index Description      Extractor           Ordered
=================================================================== 0
        No index found .getFirstName()     false
1       No index found .getAge()           false
```

7 - 49

In this example, a query was performed on a cache against two fields `firstName` and `Age`, neither of which is indexed. The resultant costs come from the number of rows touched multiplied by a constant. In this case, 23000 rows existed, which were multiplied by the relative weight of 1000. Because neither field was indexed, the index lookup selection lists the extractor used to get the value and no index type. If an index were present, the result would have shown the index type, for example, `SimpleMapIndex`. A more in-depth analysis of a query using index and type could be performed to determine the value of the index itself by examining the cost of the portion of the query impacted by the index and the index itself, and then a decision could be made on performance improvement versus the cost, in terms of memory of the index.

TRACE Output

TRACE output is broken up into two parts:

- TRACE – A breakdown of a query showing index, effectiveness (measure of impacted rows), and duration

```
Name                                  Index    Effectiveness    Duration
=========================================================================
com.tangosol.util.filter.AndFilter | ----   | 12852 |1386(89%) | 134
   LikeFilter(.getFirstName(), Al%)| 0      | 12852|2142(83%)  | 85
   com.tangosol.util.filter.And…   | ----   | 2142|1386(35%)   | 42
     GreaterFilter(.getAge(), 16)  | 1      | 2142|2058(3%)    | 17
     LessFilter(.getAge(), 50)     | 1      | 2058|1386(32%)   | 24
```

- Index – Associated index and extractor

```
Index Lookups
Index Description     Extractor                                  Ordered
=========================================================================
0     SimpleMapIndex: Extractor=.getFirstName( .getFirstName() true
. . .
```

7 - 50

Trace output is similar to explain plan output, except that each row represents the actual results of the query rather than an estimate.

Columns are:

- **Name** and **Index** – The name of the filter in the query and its index
- **Effectiveness** – This column shows the amount by which a key set was actually reduced as a result of each filter. The value is given as before filter **key set size** | **after filter key set_size** and is also presented as a percentage. The **before** value represents the key set size before evaluating a filter or applying an index. The **after** value represents the size of the key set remaining after evaluating the filter or applying the index. For a composite filter entry, the value is the overall result for its contained filters. When a key set size can no longer be reduced based on an index, the resulting key set is deserialized and any non-index filters are applied.
- **Duration** – This column shows the number of milliseconds spent evaluating the filter or applying an index. A value of `0` indicates that the time registered was below the reporting threshold.

TRACE Output: Example

Example Trace Output

```
"firstName like 'Al%' and (age > 16 and age < 50)"

...
cache.addIndex(new ReflectionExtractor("getAge"), true, null);
cache.addIndex(new ReflectionExtractor("getFirstName"), true, null);
...
QueryRecorder agent = new QueryRecorder(RecordType.TRACE);
...

Name                                  Index    Effectiveness    Duration
=========================================================================
com.tangosol.util.filter.AndFilter | ----   | 12852 |1386(89%) | 134
   LikeFilter(.getFirstName(), Al%)| 0      | 12852|2142(83%)  | 85
   com.tangosol.util.filter.And…   | ----   | 2142|1386(35%)   | 42
     GreaterFilter(.getAge(), 16)  | 1      | 2142|2058(3%)    | 17
     LessFilter(.getAge(), 50)     | 1      | 2058|1386(32%)   | 24


Index Lookups
Index Description     Extractor                                 Ordered
=========================================================================
0     SimpleMapIndex: Extractor=.getFirstName( .getFirstName() true
1     SimpleMapIndex: Extractor=.getAge(), Ord .getAge()       true
```

7 - 51

In this example, the exact same query was performed; however, two indexes were added, on first name and age. The example shows that the greatest impact was achieved by the first filter, which reduced the entire key set from 12000 to roughly 2000, or 83%. The remaining filters had minimal impact. An interesting experiment would be to apply the same query with second index and examine the overall duration versus memory use for the index itself.

Using QueryPlus with TRACE and EXPLAIN PLAN

QueryPlus can be completed with TRACE and EXPLAIN PLAN to examine query performance.

Start QueryPlus

```
/opt/coherence/bin > query.sh
```

Prefix `select` with `trace` to perform trace

```
CohQL>  trace
          select FirstName,LastName,Age
            from "dist-person"
               where age > 15 and age < 50 and FirstName like "Al%";
…
```

Prefix `select` with `explain plan for` to execute explain plan

```
CohQL > explain plan for
          select FirstName,LastName,Age
            from "dist-person"
              where age > 15 and age < 50 and FirstName like "Al%"
```

Review: Add index

```
CohQL> create index "dist-person" FirstName

. . .
```

Review: Remove index

```
CohQL> drop index "dist-person" FirstName
```

7 - 52

QueryPlus can be used to interactively execute trace and explain plan statements. Use of these two features is simple and straightforward.

1. Execute the QueryPlus tool by using the query. [sh|cmd] script.
2. To execute a trace statement simply prefix the select statement with the trace keyword. The result of the trace operation will be displayed.
3. To execute an explain plan, prefix a select with the keywords explain plan for.
4. Experiment with trace and explain plan by adding and removing indexes as shown in the example.

Summary

In this lesson, you should have learned how to:

- Develop Filters to subset cache data, including sorting and aggregation of data objects
- Construct Filters using ValueExtractors
- Develop custom aggregators, which can run in parallel in a Coherence cluster
- Extend objects to support data affinity
- Implement Java clients that can perform dynamic queries using the Coherence Query Language (CohQL)
- Improve filter and query performance using indexes, trace and explain plans