Workday OMS Programming Exercise

Efficient Range Queries

Workday applications are built on a platform that supports large volumes of data stored in a distributed in-memory database. We would like to enable our app developers to ask questions like:

```
Find workers Where salary > 10 and salary < 20
```

It's obviously important to get answers quickly. For Workers this isn't a big deal since even the largest companies only have 10^5 Workers. It becomes a little more interesting when you talk about higher volume cases, such as:

```
Find PayrollResult
where amount > 10,000
  and date < 1/2/13
  and location='China' ...
```

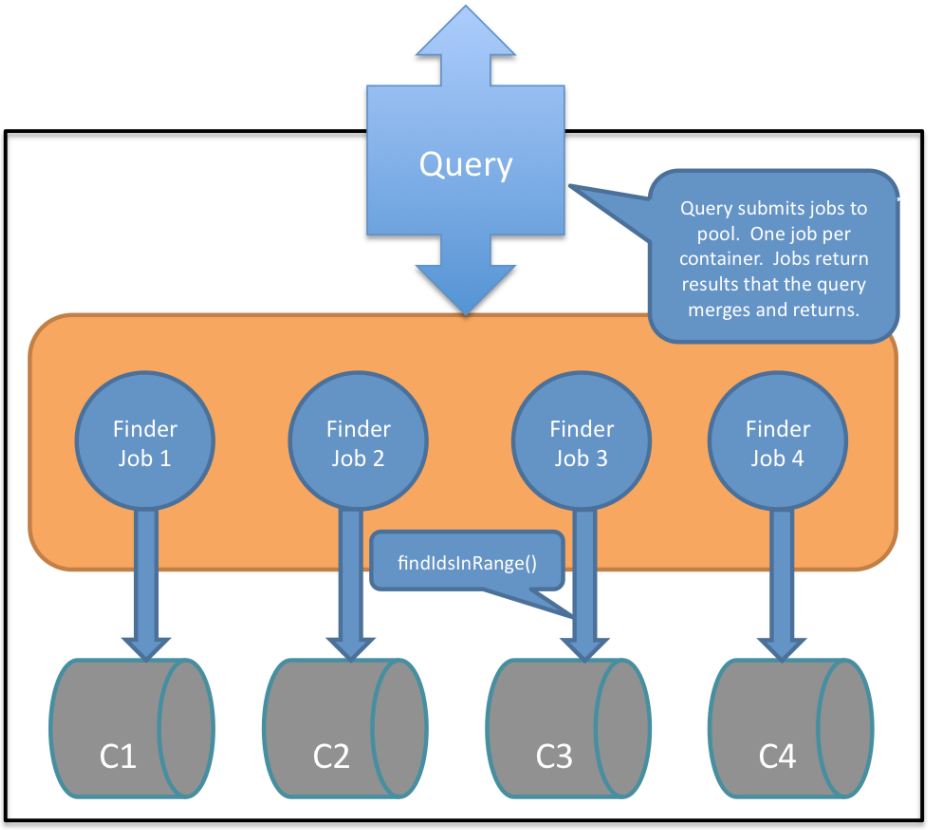
We expect to have millions and billions of PayResults.

One way to approach this is to have developers create records or documents that describe each instance of Worker or PayrollResult and store these documents efficiently to our in-memory database. Here are some examples:

```
Worker {
  name:"Bob Jones"
  salary:1,000,000
  location:Erehwon
}

PayrollResult {
  worker: 123456
  net: $1002.34
  date: 12/31/2011
}
```

To support (amongst other things) the parallelization of queries on this database we will partition the data into containers/shards.



A typical query request, such as Find payResultLines where net > 10,000 and net < 100,000 would be processed as so:

1. A query request is dispatched to a computation node that houses several containers.
2. The computation node in turn dispatches a job into a fork/join pool. Each job invokes findIdsInRange(10,000, 100,000, false, false) on a container.
3. The jobs are executed with results joined together by the query processor and returned to caller.

Requirements:

Implement a Container to hold numeric values (like 'net' above), and answer basic range queries. The API to implement is simple:

```
interface RangeContainerFactory {
  /**
   * builds an immutable container optimized for range queries.
   * Data is expected to be 32k items or less.
   * The position in the "data" array represents the "id" for that instance
   * in question. For the "PayrollResult" example before, the "id" might be
   * the worker's employee number, the data value is the corresponding
   * net pay. E.g, data[5]=2000 means that employee #6 has net pay of 2000.
   */
  RangeContainer createContainer(long[] data);
}

/**
 * a specialized container of records optimized for efficient range queries
 * on an attribute of the data.
 */
interface RangeContainer {

  /**
   * @return the Ids of all instances found in the container that
   * have data value between fromValue and toValue with optional
   * inclusivity. The ids should be returned in ascending order when retrieved
   * using nextId().
   */
  Ids findIdsInRange(long fromValue,
                     long toValue,
                     boolean fromInclusive,
                     boolean toInclusive);
}

/**
 * an iterator of Ids
 */
interface Ids {
  /**
   * return the next id in sequence, -1 if at end of data.
   * The ids should be in sorted order (from lower to higher) to facilitate
   * the query distribution into multiple containers.
   */

  static final short END_OF_IDS = -1;

  short nextId();
}
```

Validation:

Your implementation should pass the following simple JUnit test before you start tuning for performance with volume data:

```
public class RangeQueryBasicTest {
  private  RangeContainer rc;

  @Before
  public void setUp(){
    RangeContainerFactory rf = new YourRangeContainerFactory();
    rc = rf.createContainer(new long[]{10,12,17,21,2,15,16});
  }

  @Test
  public void runARangeQuery(){
        Ids ids = rc.findIdsInRange(14, 17, true, true);
        assertEquals(2, ids.nextId());
        assertEquals(5, ids.nextId());
        assertEquals(6, ids.nextId());
        assertEquals(Ids.END_OF_IDS, ids.nextId());

        ids = rc.findIdsInRange(14, 17, true, false);
        assertEquals(5, ids.nextId());
        assertEquals(6, ids.nextId());
        assertEquals(Ids.END_OF_IDS, ids.nextId());

        ids = rc.findIdsInRange(20, Long.MAX_VALUE, false, true);
        assertEquals(3, ids.nextId());
        assertEquals(Ids.END_OF_IDS, ids.nextId());
    }
}
```

FAQ

Can't we just use a B-Tree/ NavigableMap/HashTable/BinarySearch/Lucene...

Sure, if you want. But there are a number of things we'd like to achieve:

- The container will be in memory and will need to hold lots of data, so we'd like it to be as space efficient as possible, but not to the extent where speed is significantly compromised. Speed is king, memory/disk can be purchased.
- Since we hate collecting garbage, we need to be careful about memory allocation during a query, as my grandmother told me, "allocate not, gc not".
- Code should be clear and understandable.
- **The rough order of trade off to consider for this case is: search performance, efficient usage of memory, simplicity and maintainability of the code**

These objectives are often at odds - we'd like to explore what's possible.

What about Big O? Should it be O(log n) or O(log log n)?
We partition the data into Containers of 32K instances each. So, Big O analysis is sort of useless here, as we're not concerned about how the algorithm grows, only with how it performs when n=32K.

Test Methodology

Candidates will be expected to implement a solution to this problem at home. The implementation should be sent to Workday before the on-site interview. As part of the interview, the candidate will have a face-to-face open-ended discussion with Workday developers regarding the design and implementation. This discussion will explore design decisions and alternative implementation choices. The solution should include a JUnit test suite to benchmark the implementation and test it for correctness. The resulting algorithm will also be tested with varieties of data to see how well it performs under different conditions.

Some Minor Details

To facilitate a quick processing of your implementation, please pay attention to the following minor details.

1. Please use package name “com.workday” for all your classes.
2. Please do not directly modify the given interfaces.