Portable object constant-time field lookup

Problem

Portable object serialization is first released in Apache Ignite 1.4 as an experimental feature and is not available from public API.

Some critical Ignite features like indexing/queries require object introspection without deserialization. To achieve this portable protocol writes object fields as follows:

\[
\begin{align*}
[X..X+4] & \quad \text{Field 1 ID} \\
[X+4..X+8] & \quad \text{Field 1 length} \\
[X+8..Y] & \quad \text{Field 1 content} \\
[Y..Y+4] & \quad \text{Field 2 ID}
\end{align*}
\]

To read a field we get its ID and search for it starting from the object head. This gives good performance and data locality for regular serialization/deserialization cycles because normally fields are written in the same order as they are read. As a result usually we can find the field during read in \(O(1)\) time.

```java
void writePortable(PortableWriter w) {
  w.write("A", a);
  w.write("B", b);
}

void readPortable(PortableReader r) {
  a = r.read("A");
  b = r.read("B");
}
```

To the contrast, indexing engine usually reads fields in random order with \(O(N)\) complexity on average.

In Ignite 1.5 we need to change the protocol so that field position could be found in \(\sim O(1)\) on average.

Protocol

The following changes are proposed:

- Field lengths are replaced with relative offsets;
- Field IDs and offsets are moved to object footer;
- Relative footer offset is added to object header;
- All field IDs are hashed in order they are written. Resulting value is saved to object header. We refer to is as **schema ID**.

Resulting object layout (unrelated header pieces are omitted):

\[
\begin{align*}
[0..4] & \quad \text{Footer offset; could be zero if the whole object is written in raw mode.} \\
[4..8] & \quad \text{Schema ID. Absent in case footer offset is zero.} \\
[8..X] & \quad \text{Field 1.} \\
[X..Y] & \quad \text{Field 2.} \\
[Y..Y+4] & \quad \text{Footer start, field 1 ID.} \\
[Y+4..Y+8] & \quad \text{Field 1 offset.} \\
[Y+8..Y+12] & \quad \text{Field 2 ID, etc.}
\end{align*}
\]

Implementation details

Object schema

We define each unique set of written fields as **schema**. The following example demonstrates two schemas:
void writePortable(PortableWriter w) {
    w.write("A", a);
    w.write("B", b);
    if (b)
        w.write("C", c); // Schema 1: [A->B->C];
    else
        w.write("D", d); // Schema 2: [A->B->D];
}

Each schema consists of:

- Schema ID which is a hash of all field IDs. E.g. ID = HASH("C" + HASH("B" + HASH("A")));
- Total fields count;
- Ordered collection of field IDs.

Known schemas are stored in read-only structure. If new schema is detected during read or write, it is updated atomically. Normally object will have only 1 schema, 2-3 schemas in rare cases, >3 schemas in very rare cases. For this reason we can store them in volatile array or so.

Schemas are stored inside existing type descriptor. This way we avoid additional hash map lookups.

Object write

- We save object field IDs and offsets to array during write. When all fields are written, we simply copy this array to object footer in a single System.
arrayCopy() or Unsafe.copyMemory() operation. To minimize GC garbage we can use thread-local array instead of "new byte[]";
- If new schema is detected at the end of object write, it is added to the list of known object schemas.
- If collision is detected on (schema ID, fields count) pair, we throw an exception in the same way as if we had field ID conflict. This should be very unlikely event. N.B.: we can support collisions with some additional computational overhead.

No or almost no additional overhead is expected comparing to Ignite 1.4 after warmup.

Object read

- Check if object’s schema is known. Normally this will require only 1-2 int comparisons. If no, we scan the whole object, create the schema and save it.
- Until object read schema matches object write schema, we just read the object sequentially. Schema matching is performed using trivial int field ID comparisons.
- If read/write schemas mismatch is detected, we fallback to random field read. Normally mismatch will only occur if different object versions co-exist in runtime.

No or almost no additional overhead is expected comparing to Ignite 1.4 after warmup.

Random object field read

- Check if object’s schema is known. Normally this will require only 1-2 int comparisons. If no, we scan the whole object, create the schema and save it.
- Field name is converted to field ID as usual;
- Schema is queried for field ID order. We need to evaluate possible techniques for fast int lookup: normal HashMap, open-addressing like in ThreadLocal’s, specialized int-int maps. Anyways, even HashMap should usually sustain O(1) complexity;
- Go to footer and get field offset: FieldOffset = INT_VALUE_AT(ObjectStart + FooterOffset + FieldIdOrder * 8 + 4);
- Use FieldOffset to get the field.