Status

This proposal is currently on hold as it seems most users want the transaction to fail if the mutation doesn't have an effect.

Semantics

The `compare_and_set` mutation would consist of a key, an expected former value, and a value to set the key to if the former value matches the expected former value. The expected former value can either be absent (meaning the key is not set) or a value to be compared with the former value. Likewise, the value to set if the comparison succeeds can be absent (indicating that the key should be cleared), or present (indicating the value should be updated). This way it can be used to set a key to a value only if it was previously absent.

Caveats

Like other mutations, the transaction succeeds or fails solely based on the read conflict range of that transaction. The transaction committing successfully does not tell you any information about what the former value of the key was. Similarly, each of these mutations in a transaction would behave independently. For example, a scheme that attempts to swap the values of two keys using two of these mutations would not be sound, since it could be that only one of these mutations has an effect.

Examples

```
Key: "foo", ValueToCompare: "old", ValueToSet: "new"
```

If the key "foo" is currently set to "old", then after applying this mutation the value would be "new". Otherwise this mutation would have no effect.

```
Key: "foo", ValueToCompare: <absent>, ValueToSet: "new"
```

If the key "foo" is currently not set, then after applying this mutation the value would be "new". Otherwise this mutation would have no effect.

```
Key: "foo", ValueToCompare: "old", ValueToSet: <absent>
```

If the key "foo" is currently set to "old", then after applying this mutation the key "foo" would not be present. Otherwise this mutation would have no effect.

Proposed c api

```
DLLEXPORT void fdb_transaction_compare_and_set(FDBTransaction* tr,
                                               uint8_t const* key_name,
                                               int key_name_length,
                                               uint8_t const* former_value,
                                               int former_value_length,
                                            fdb_bool_t former_value_present,
                                         uint8_t const* potential_new_value,
                                             int potential_new_value_length,
                                    fdb_bool_t potential_new_value_present);
```

Changes to MutationRef serialization

MutationRef is currently a typecode and two arbitrary bytestring payloads. We can keep this scheme, and match other atomic ops by using the first string payload as the key. The second bytestring payload can be serialized pair of optional values. The format can be an implementation detail. I think we can use either BinaryWriter or ObjectSerializer. To be more "future proof" we can consider using the ObjectSerializer, but I don't anticipate us evolving the schema.

Protocol version compatibility

We may want to introduce this in a new protocol version so that we can be sure that the server understands it if the client sends these mutations.

Downgrade considerations

In order to support downgrade, we would need to teach the previous protocol version how to read and interpret this mutation, but we should not allow clients at the previous protocol version to write this mutation. This is to avoid an old server on that protocol version not understanding a mutation from a new client on that protocol version. This allows us to add the ability to downgrade to an already-existing protocol version.

Backup compatibility implications

Presumably backups taken on the new version with this feature would not be possible to apply to with older versions.

Implementation details

WIP

RYW cache

Storage server

Storage cache

Other places that need to know how to interpret mutations?

Test plan

1. Test combining this feature with backup and dr to make sure that applying a backup still works.
2. Add this to the `FuzzApiCorrectness` workload.
3. Add to binding tester.
4. Test downgrades with a workload that uses this mutation.
5. Add this to the `AtomicOpsApiCorrectness` workload.