RAMCloud Design Review

Indexing

Ryan Stutsman

April 1, 2010

Introduction

- **Should RAMCloud provide indexing?**
  - Leave indexes to client-side using transactions?
- **Many apps have similar indexing needs**
  - Hash indexes, B+Trees, etc.
  - Can reduce app visible latency for indexes by optimizing server-side

Implementation Issues

- **Indexing on “opaque” data**
- **Splitting Indexes**
- **Consistency**
- **Recovery/Availability of Indexes**

Explicit Search Keys

- **Problem: RAMCloud treats objects as opaque**
  - Server-side indexing without understanding the data?

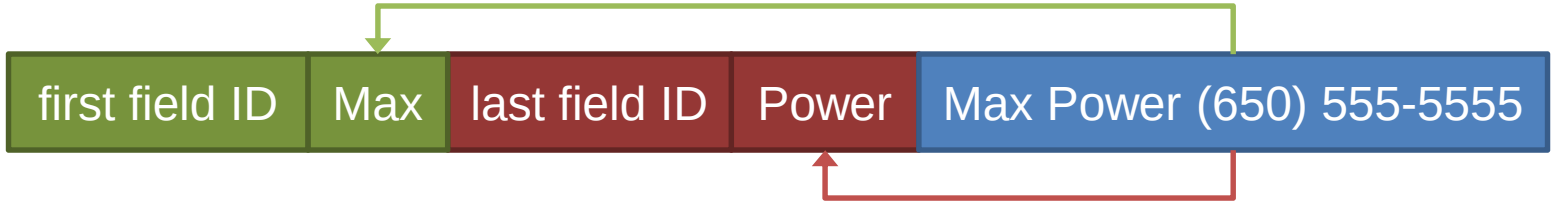
```
put(tableId, person.objectId, person.pickle())
```

Max Power (650) 555-5555

Explicit Search Keys

- **Problem: RAMCloud treats objects as opaque**
  - Server-side indexing without understanding the data?
- **Idea: Apps provide search keys explicitly**
  - Apps understand the data

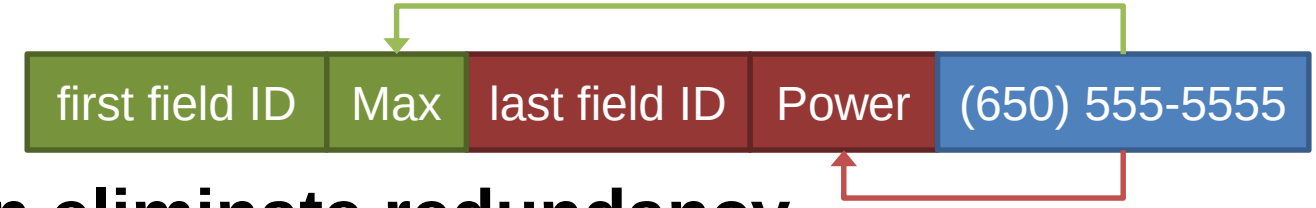
```
put(tableId, person.objectId, {'first': person.first,
                               'last': person.last},
                              person.pickle())
```



Explicit Search Keys

- **Problem: RAMCloud treats objects as opaque**
  - Server-side indexing without understanding the data?
- **Idea: Apps provide search keys explicitly**
  - Apps understand the data

```
put(tableId, person.objectId, {'first': person.first,
                               'last': person.last},
                              person.pickle())
```



- **Can eliminate redundancy**
  - Search keys need not be repeated in object
  - Search keys + Blob are returned to app on get/lookup

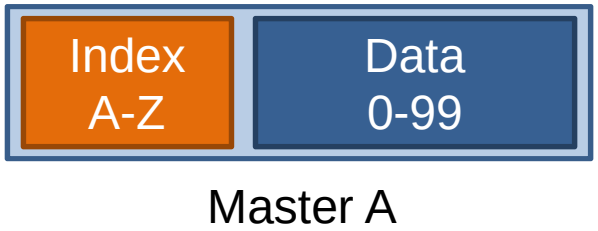
Explicit Search Keys

```
put(tableId, objectId, searchKeys, blob)

get(tableId, objectId) -> (searchKeys, blob)

lookup(tableId, indexName, searchValue) ->
                                    (searchKeys, blob)
```

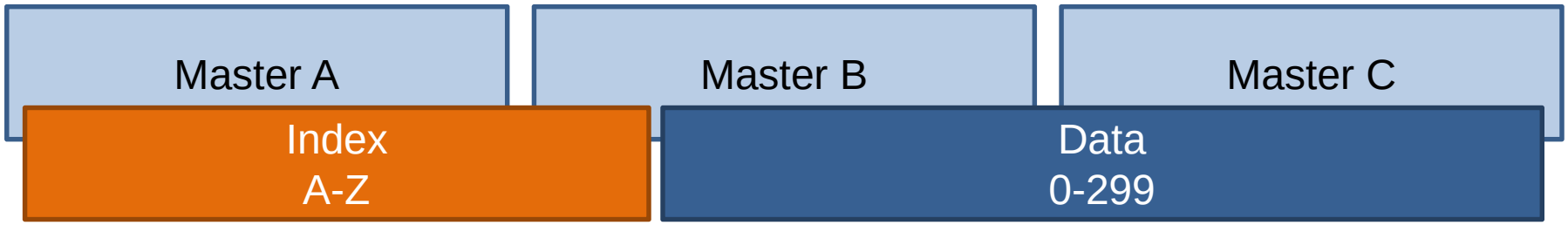
- **Put atomically updates indexes and object**
  - Details to follow

Splitting Indexes

- **Co-locate index and data**



- **Large tables?**
- **Large indexes?**
  - Can’t avoid multi-machine operations



Splitting Indexes

- **Split indexes on search key**

  - One extra access per lookup and put

- **Split indexes on object ID**

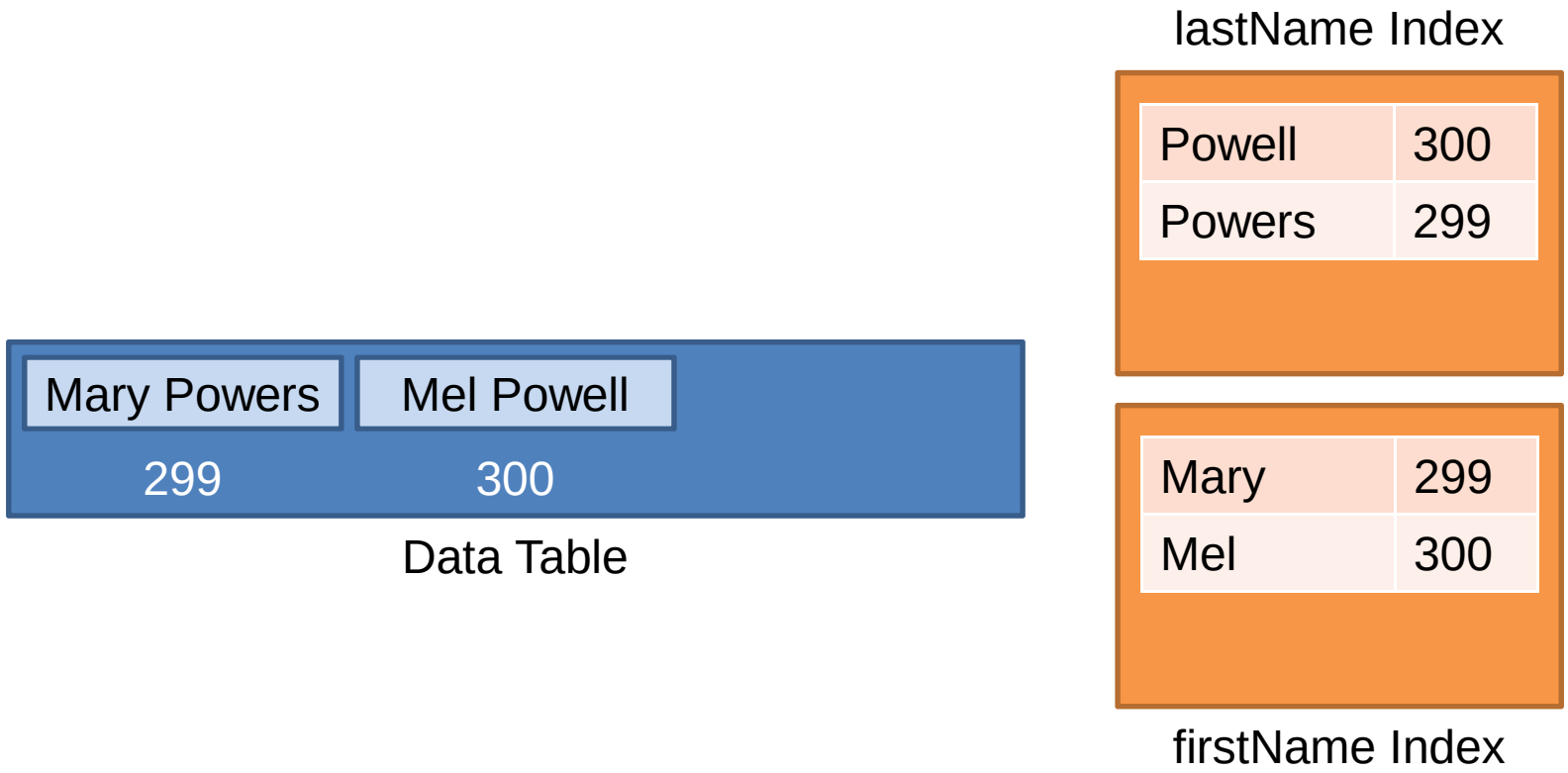
  - Lookups go to all index fragments
  - Puts are always local

- **Our decision (for now): On search key**
  - Don’t want weakest-link lookup performance

Consistency

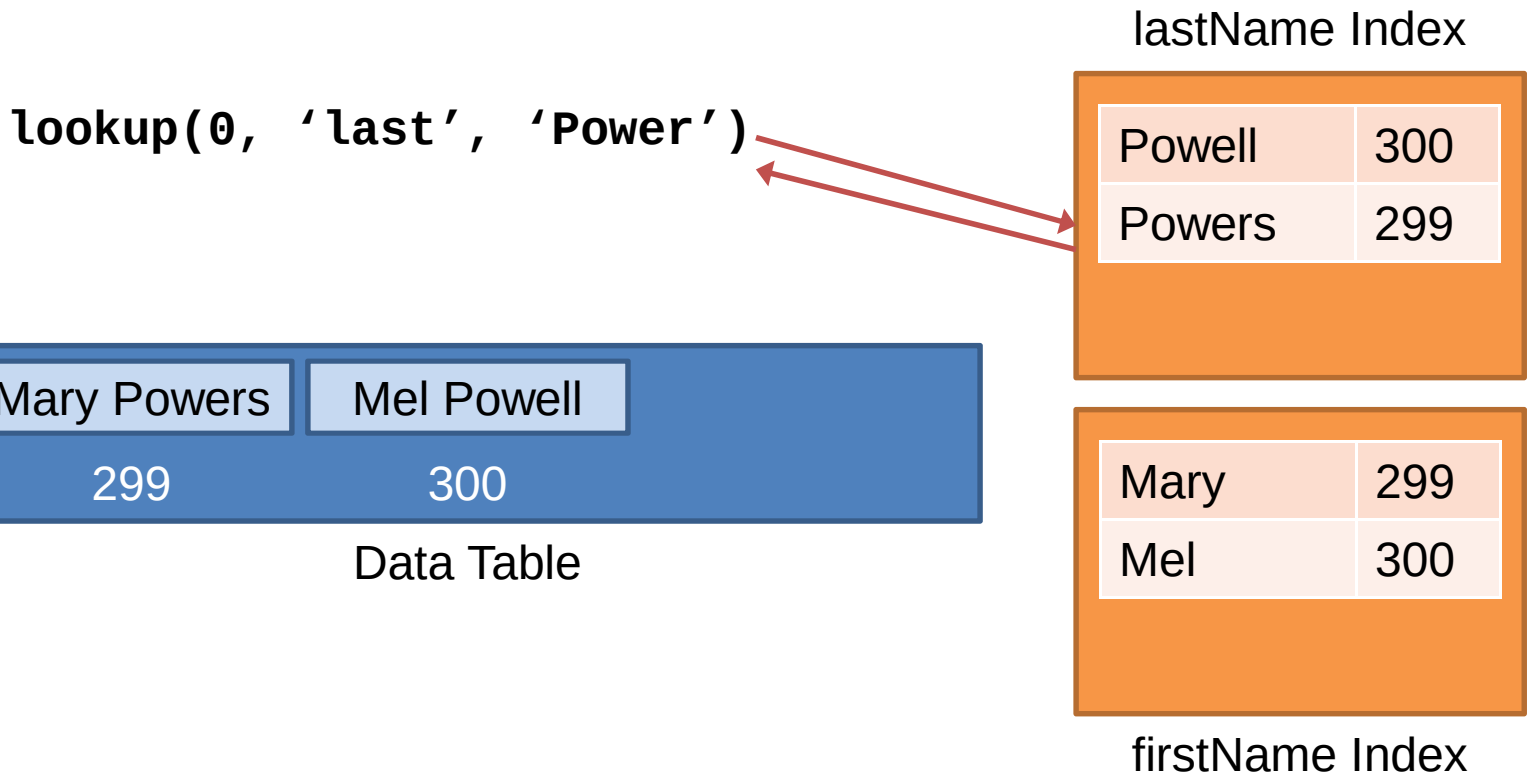
- **Problem: Index/Object inconsistency on puts**
  - Object and index may reside on different hosts
  - Apps can get objects that aren't in the index yet
  - Apps may see index entries for objects not in table yet
- **Avoid commit protocol**
- **Idea: Index entries "commit" on object put**
  - Write index entries
  - **Then** write object to table
  - Index entries considered invalid until object written
- **Turns atomic puts into atomic index updates**

Consistency



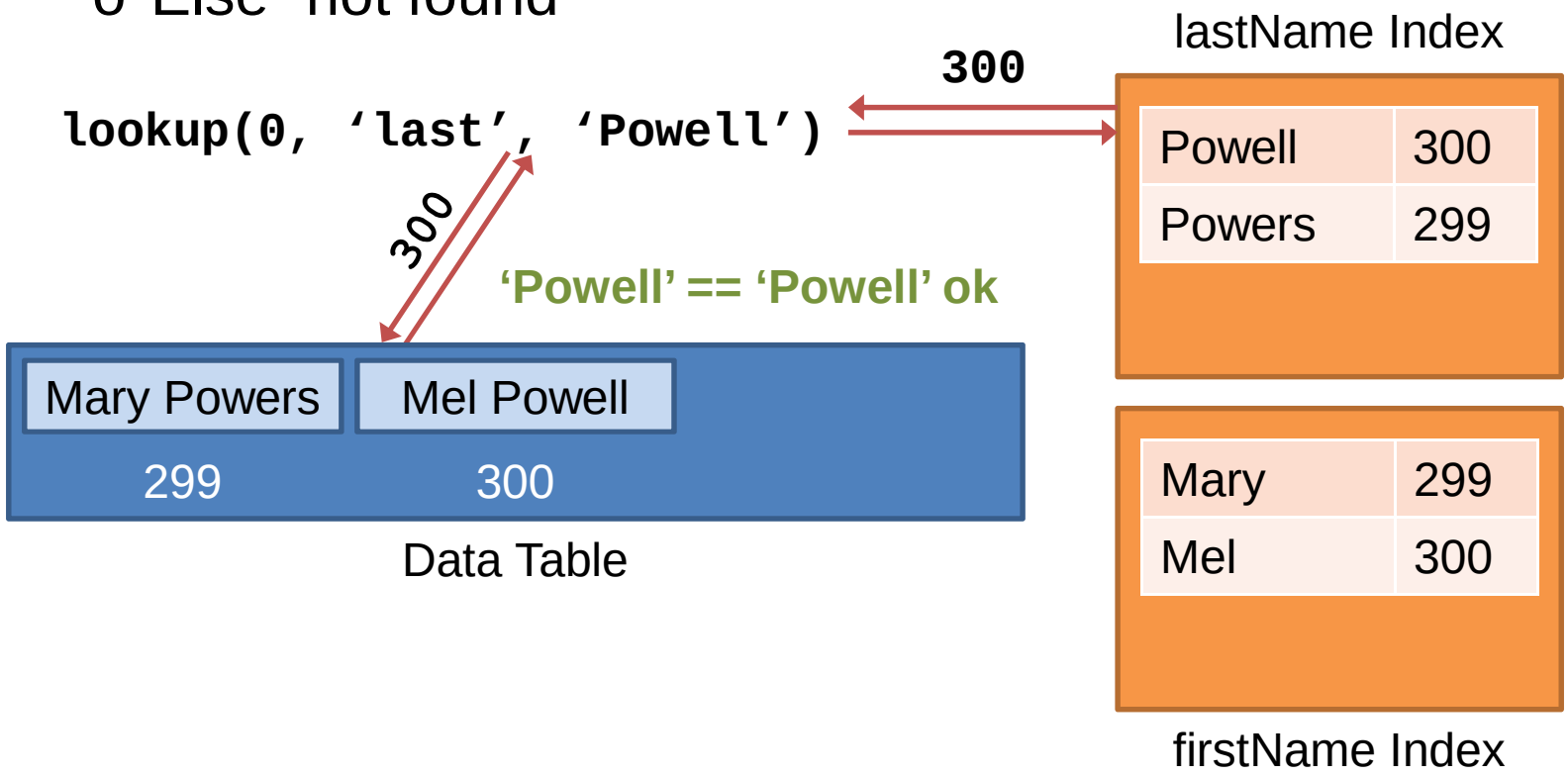
Consistency: Lookup

- **Request goes directly to correct index**
  - “Not found” returns immediately



Consistency: Lookup

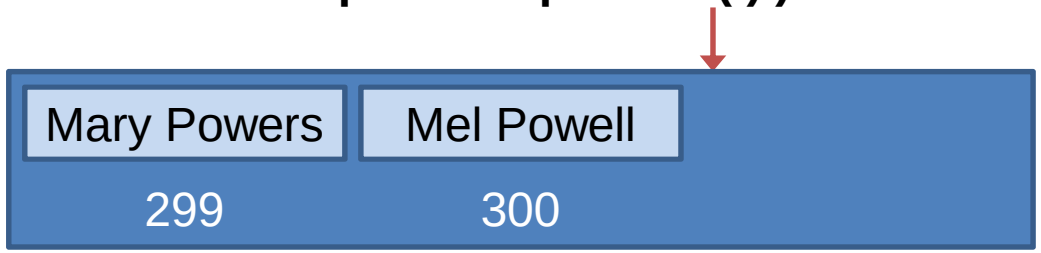
- **Consistency is checked on hit**
  - If table and index agree the return the object
  - Else “not found”



Consistency: Create

- **Insert index entries before writing object**

```
put(0, 301, {'first': 'Max',
             'last': 'Power'},
            person.pickle())
```



Data Table

lastName Index

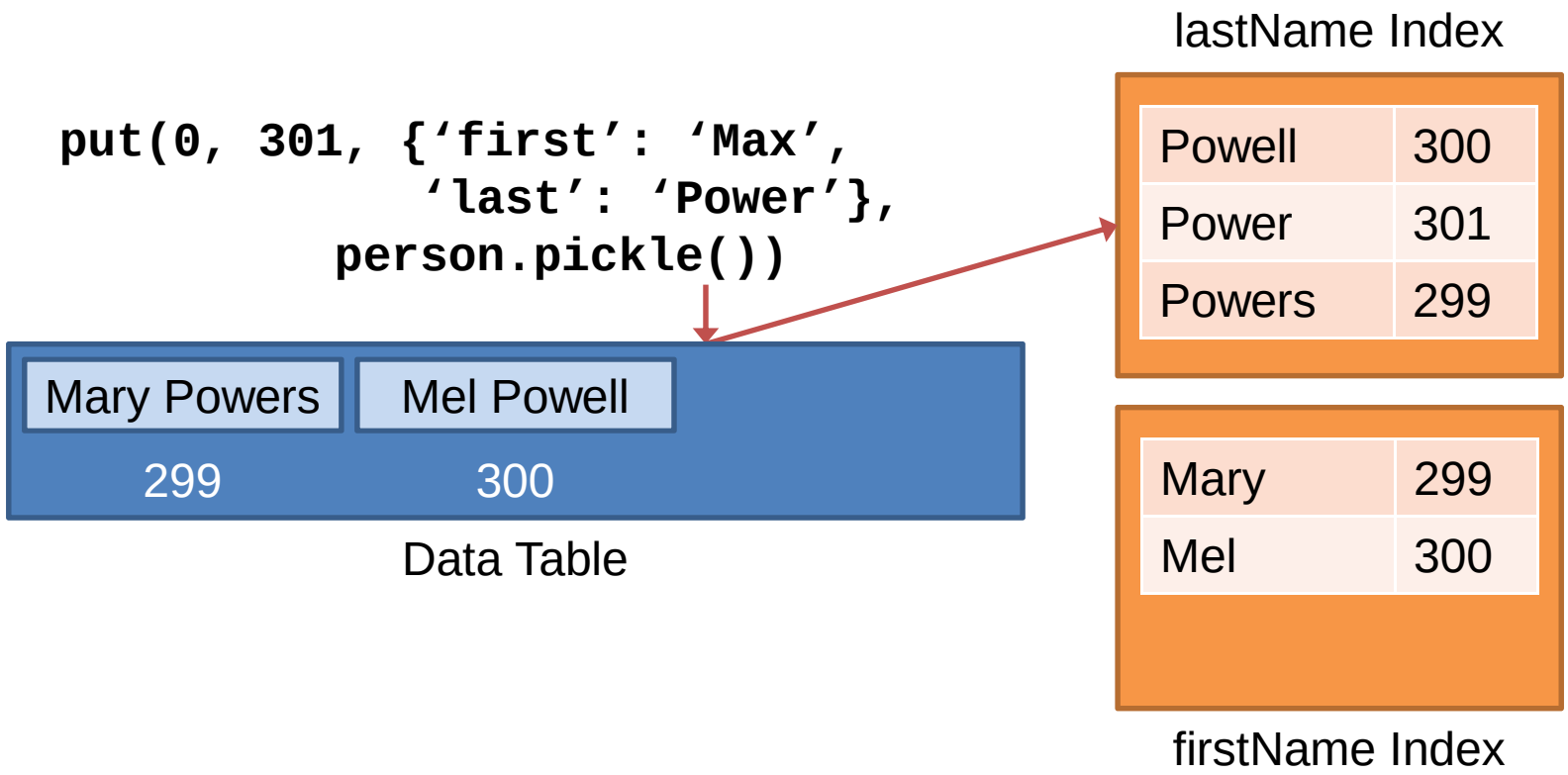
Powell	300
Powers	299

Mary	299
Mel	300

firstName Index

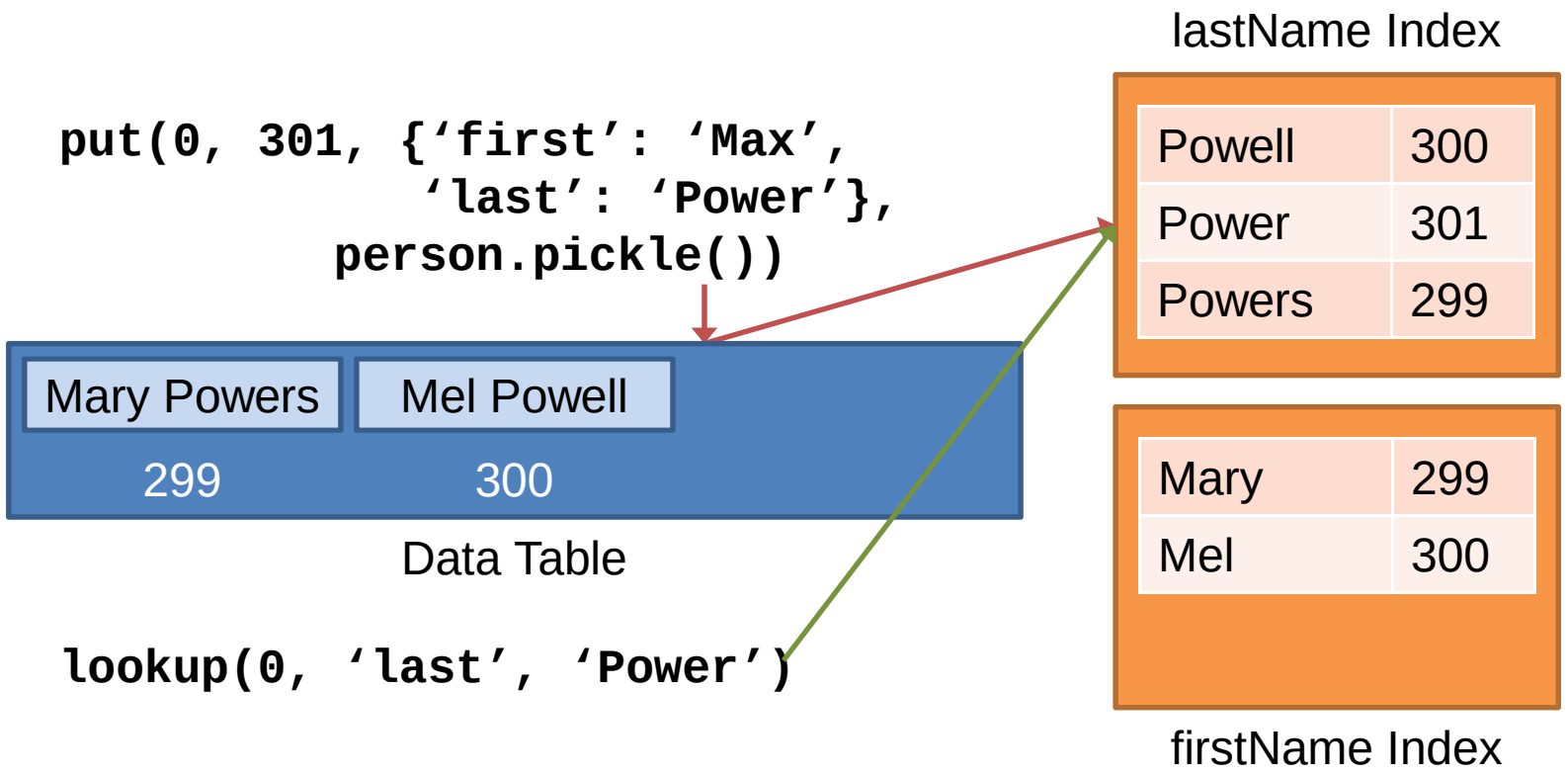
Consistency: Create

- **Insert index entries before writing object**
  - What if a lookup happens in the meantime?



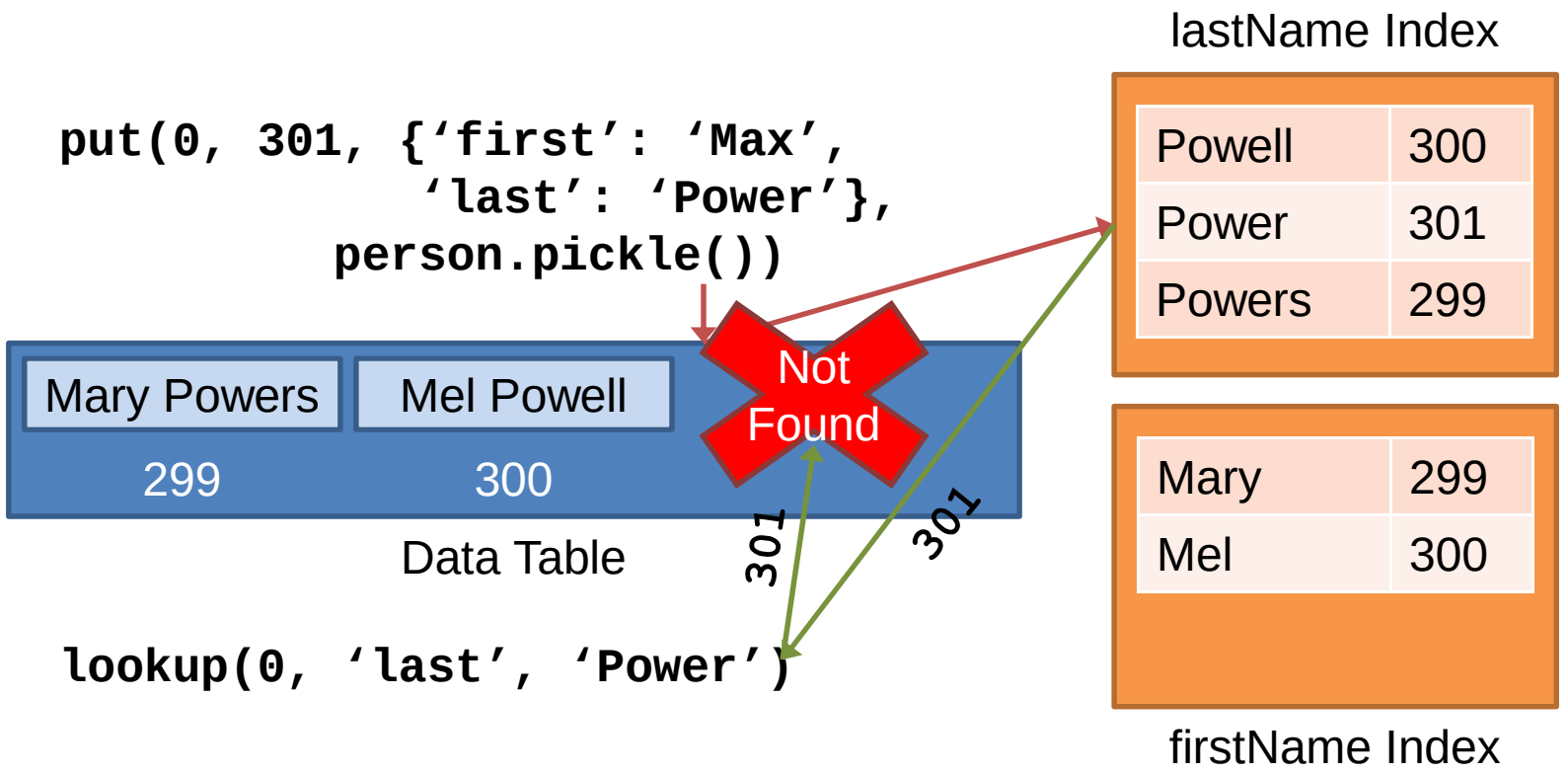
Consistency: Concurrent Lookup

- **Concurrent ops ignore inconsistent entries**



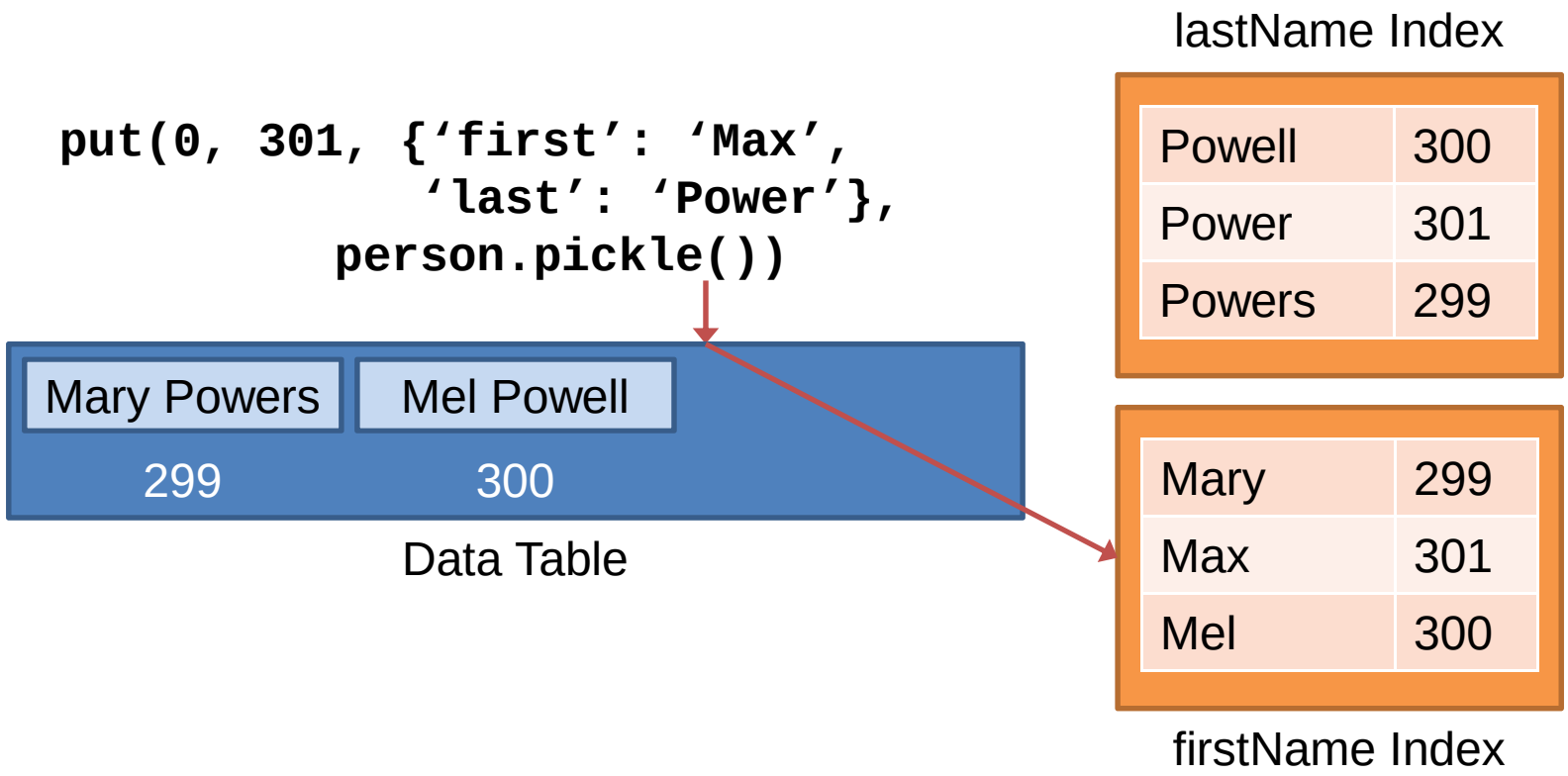
Consistency: Concurrent Lookup

- **Concurrent ops ignore inconsistent entries**



Consistency: Create (continued)

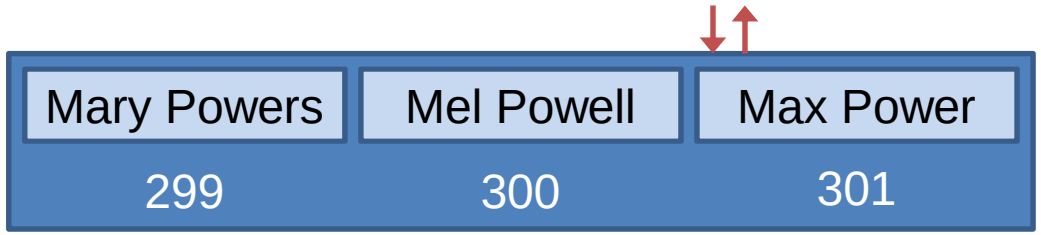
- **Insert index entries before writing object**



Consistency: Create

- **Put completes; index entries now valid**

```
put(0, 301, {'first': 'Max',
             'last': 'Power'},
            person.pickle())
```



Data Table

lastName Index

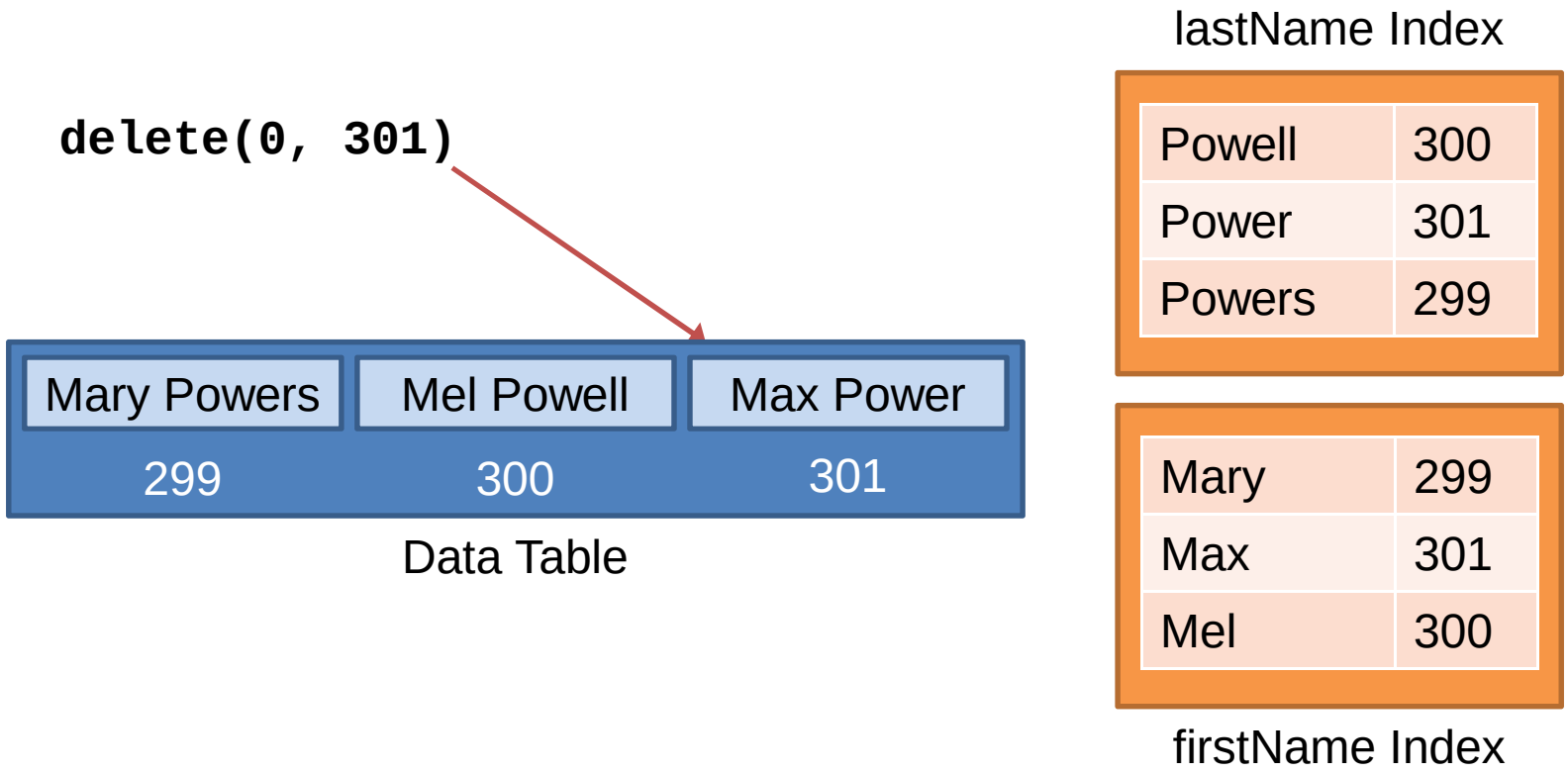
Powell	300
Power	301
Powers	299

Mary	299
Max	301
Mel	300

firstName Index

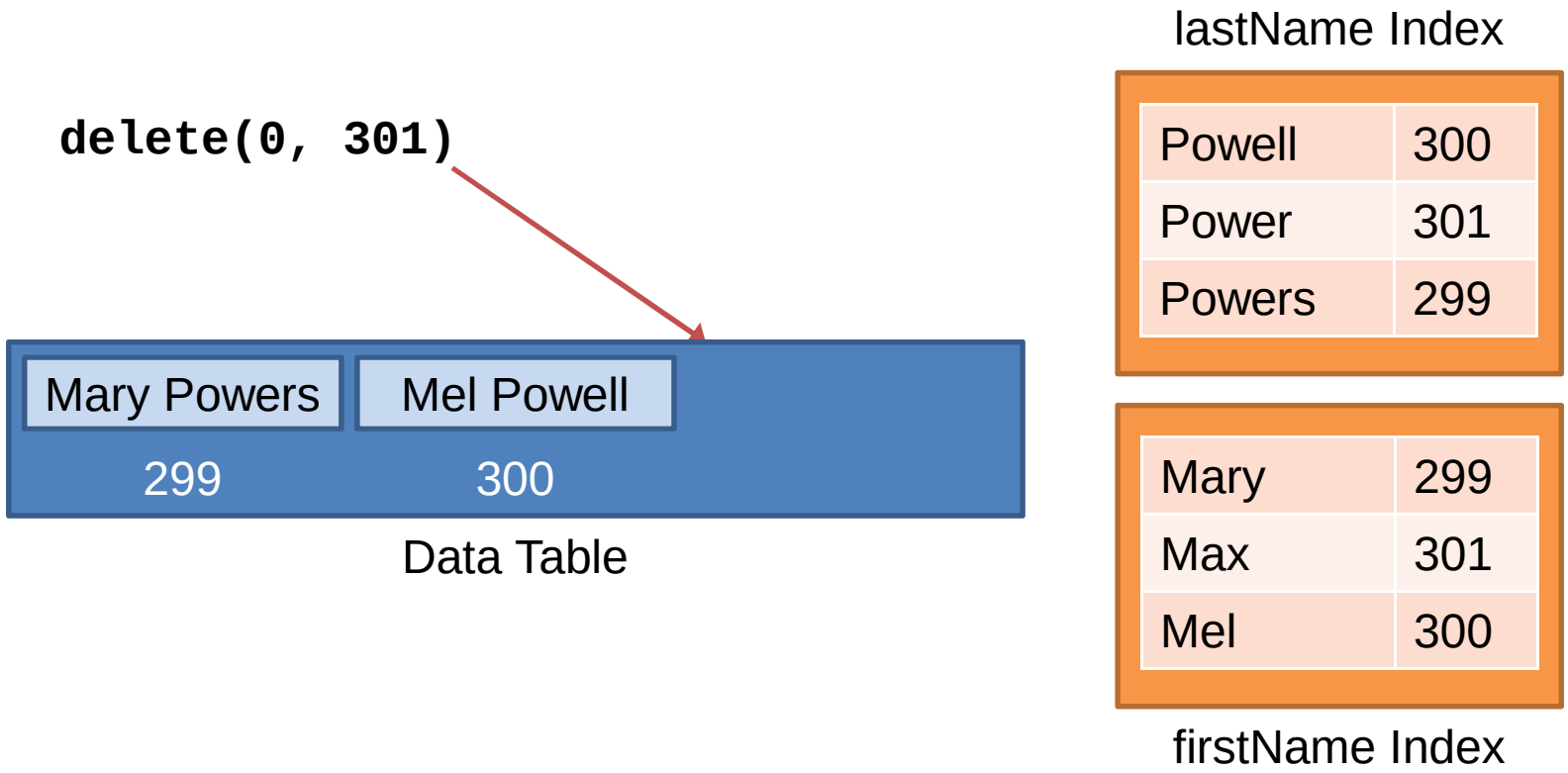
Consistency: Delete

- **Delete object first, then cleanup index entries**
  - Index entries are invalid with no corresponding object



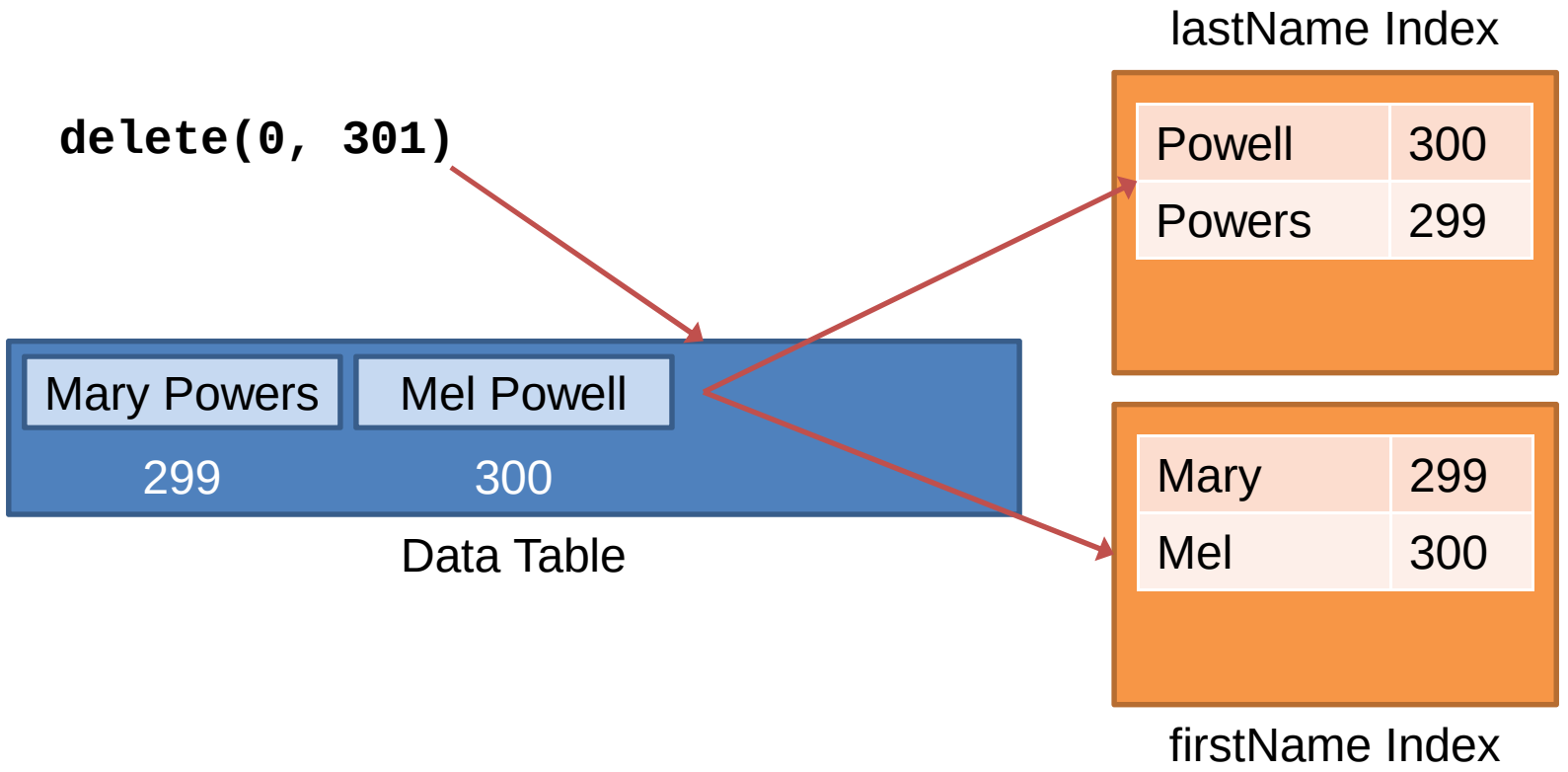
Consistency: Delete

- **Delete object first, then cleanup index entries**
  - Index entries are invalid with no corresponding object



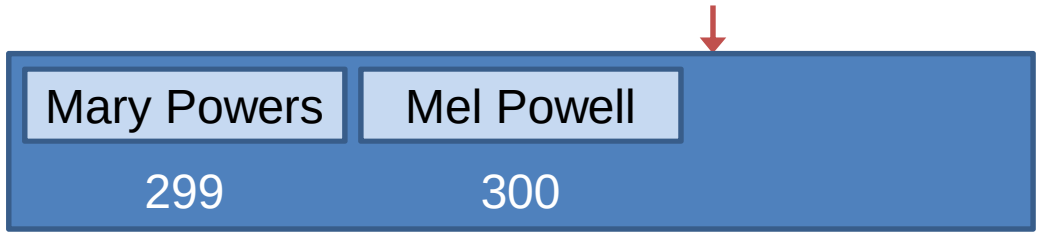
Consistency: Delete

- **Delete object first, then cleanup index entries**
  - Index entries are invalid with no corresponding object



Consistency: Update

```
put(0, 299, {‘first’: ‘Mary’,
             ‘last’: ‘Miller’},
            person.pickle())
```



Data Table

lastName Index

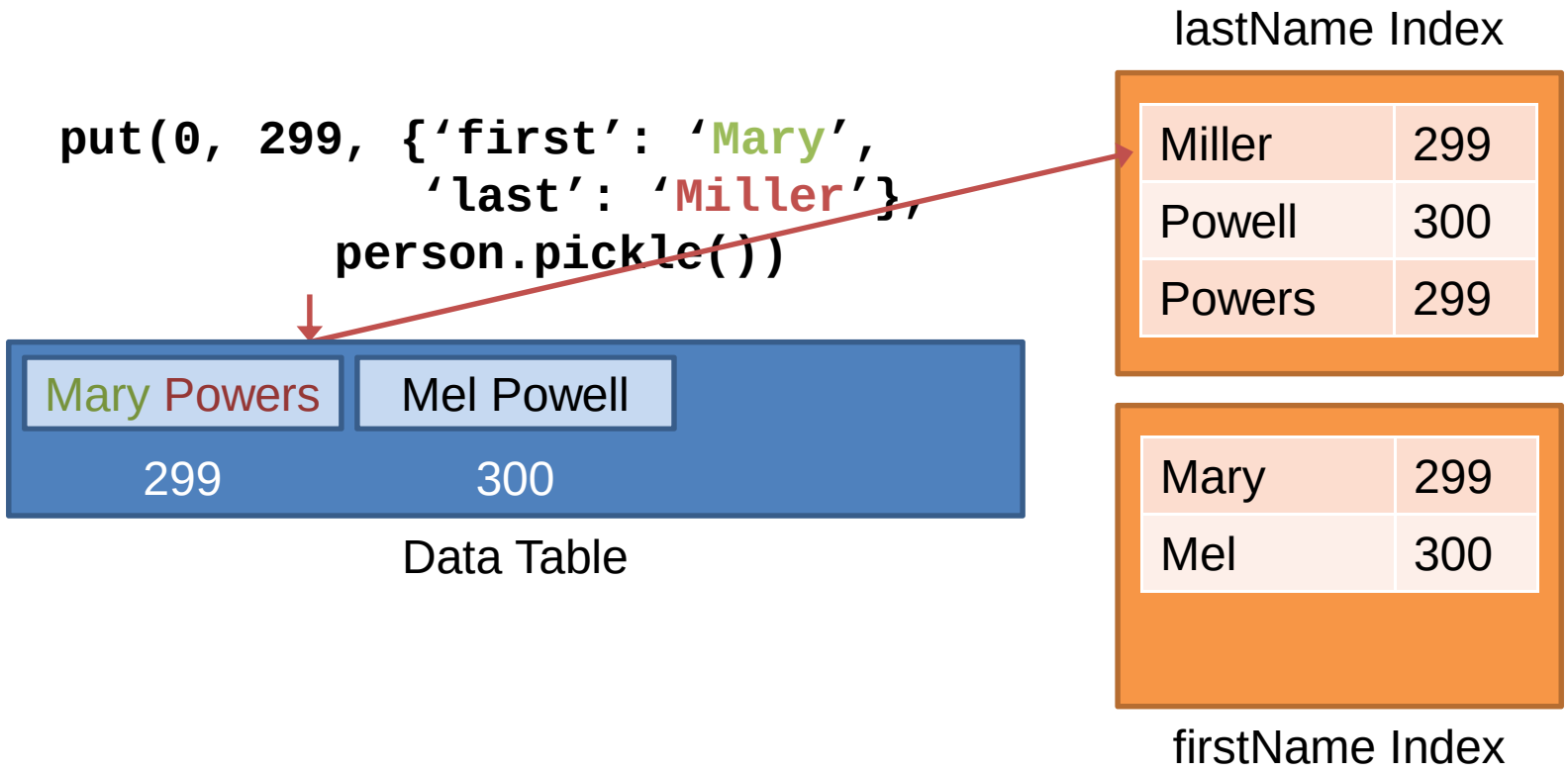
Powell	300
Powers	299

Mary	299
Mel	300

firstName Index

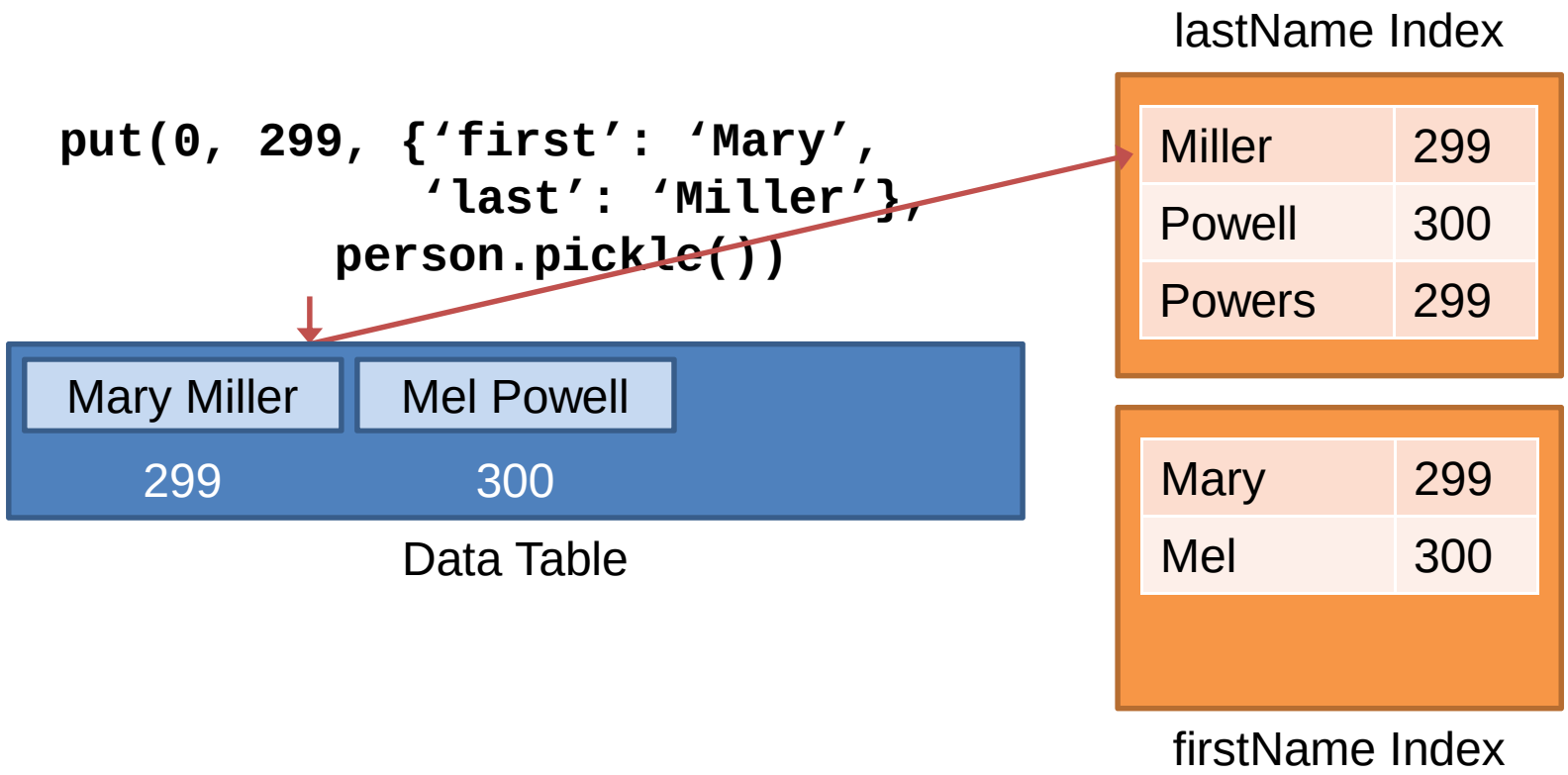
Consistency: Update

- **Compare previous index entries**
  - Insert new value if updated



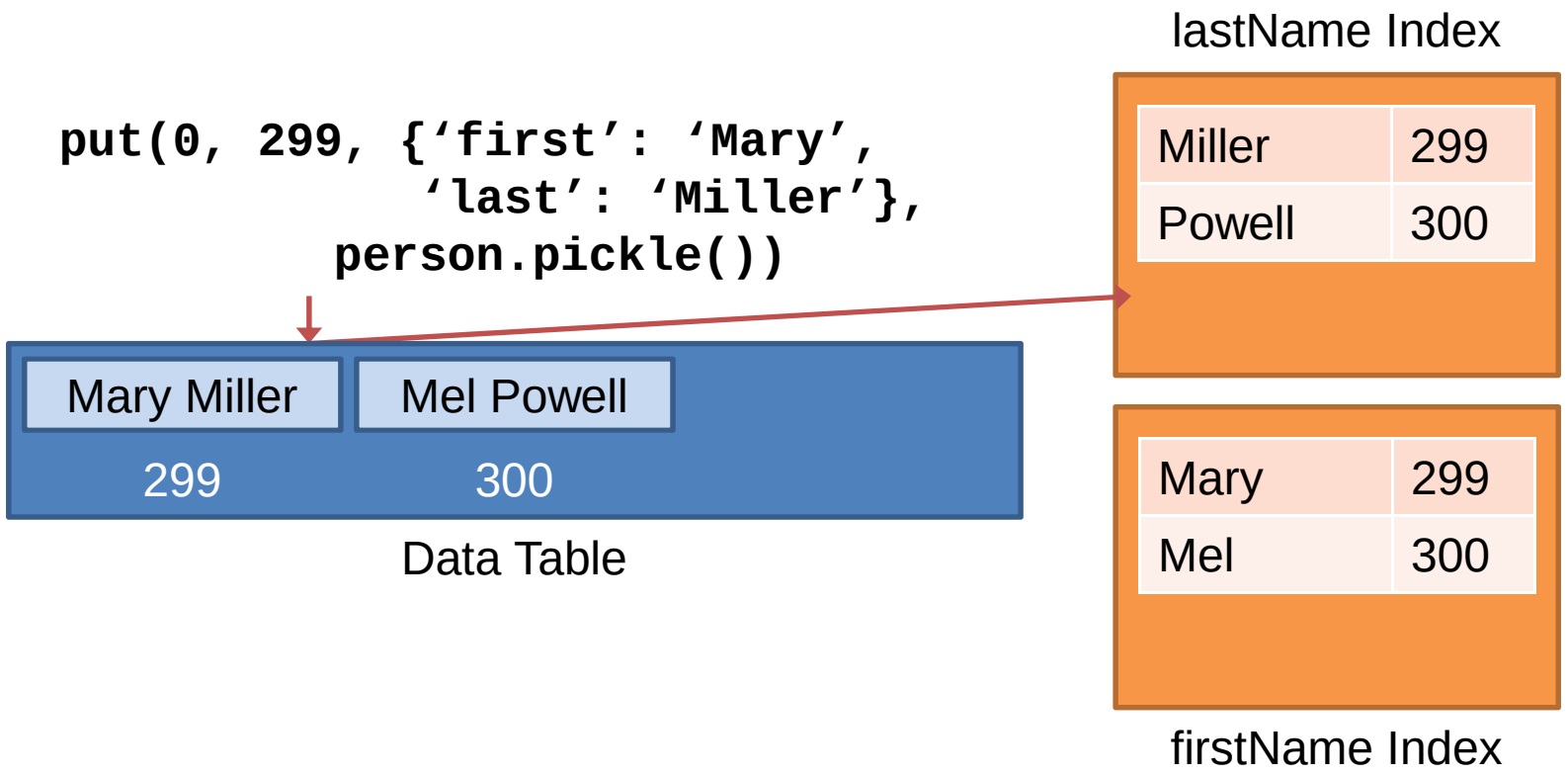
Consistency: Update

- **Commit by writing the new value**
  - Old index entries ignored by lookup since inconsistent



Consistency: Update

- **Cleanup old, inconsistent entries**



Consistency: Thoughts

- **Atomic puts give index updates atomicity**
- **Low-latency gives simplified consistency**
  - Can afford to have a single writer per object
  - Provides us with atomic put primitive for free

Index Recovery

- **Problem: Unavailable until indexes recover**
  - Many requests will be lookups
  - These will block until indexes are recovered

- **Rebuild versus Store?**
  - Storing comes at a cost to write-bandwidth
  - Possible using scale we can rebuild faster than store

Index Recovery: Partitioning

- **How far does partitioning + rebuilding get us?**
- **Worst case: Entire partition of index data only**
  - At most 640 MB
  - Larger indexes recovered a partition to a host in parallel

Index Recovery: Partitioning

Recover a single index partition on a new master:

1. **Data partitions scan, extract index entries (0.6s)**
   - Hashtable: 10 million lookups/sec
   - 640 MB / 100 byte/object = 6.4 million objects
2. **Transmit entries to new index partition (0.6s)**
   - At most 640 MB @ 10 Gbit/s
3. **New index master reinsert entries (0.6s)**
   - Similar time to master hashtable scan

- **All operations are pipelined**
  - **0.6s** to scan, extract, transmit, rebuild **total**
- **If data partitions for index in recovery add 0.6s**
  - **1.2s upper bound for conservative 100b object size**

Summary

- **Explicit search keys both flexible and efficient**
- **Split indexes on search key for fast lookup**
- **Atomic puts simplify atomic indexes**
- **Scale drives index recovery for availability**

Discussion