

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**
 - Or compose standard mechanisms to suit their needs
 - Can optimize for common needs on server-side

Implementation Issues

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

Explicit Search Keys

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

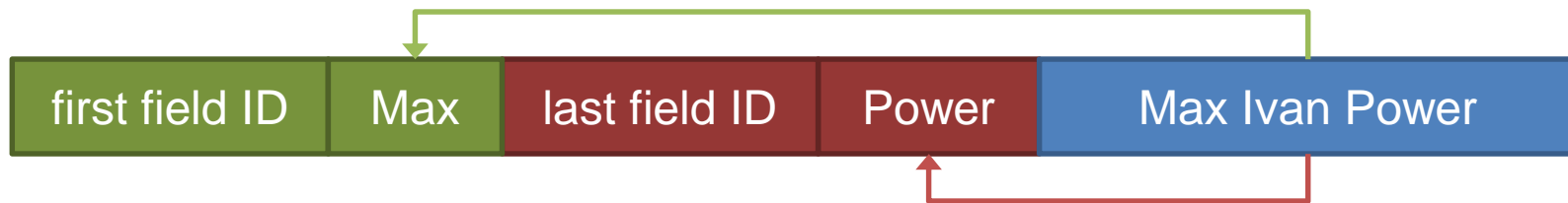
```
put(tableId, person.objectId, person.pickle())
```

Max Ivan Power

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

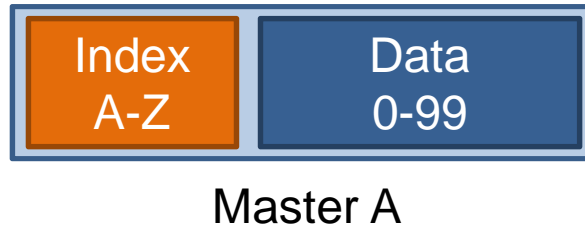
- **Lookups are distinct from gets**

```
lookup(tableId, 'last', 'Power')
```

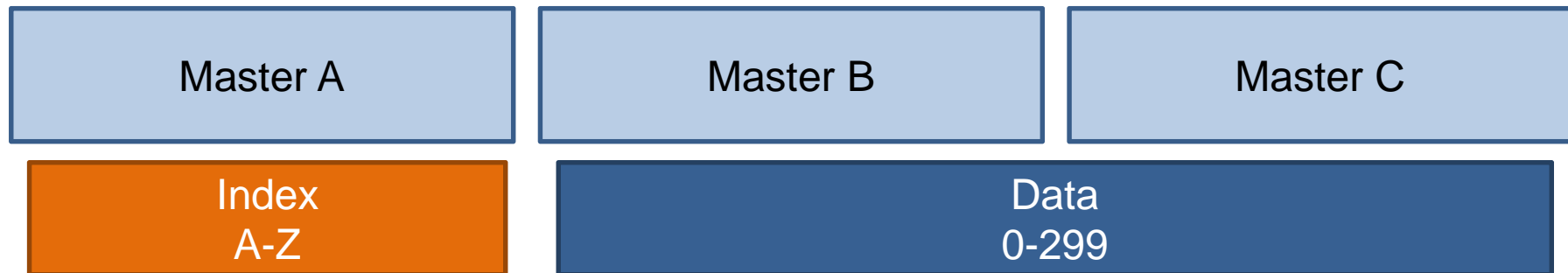
- **Put atomically updates indexes and object**
 - Details to follow

Partitioning Indexes

- **Co-locate index and data**



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



Partitioning 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
- Ordered enumeration of the index is problematic

Partitioning Indexes: Thoughts

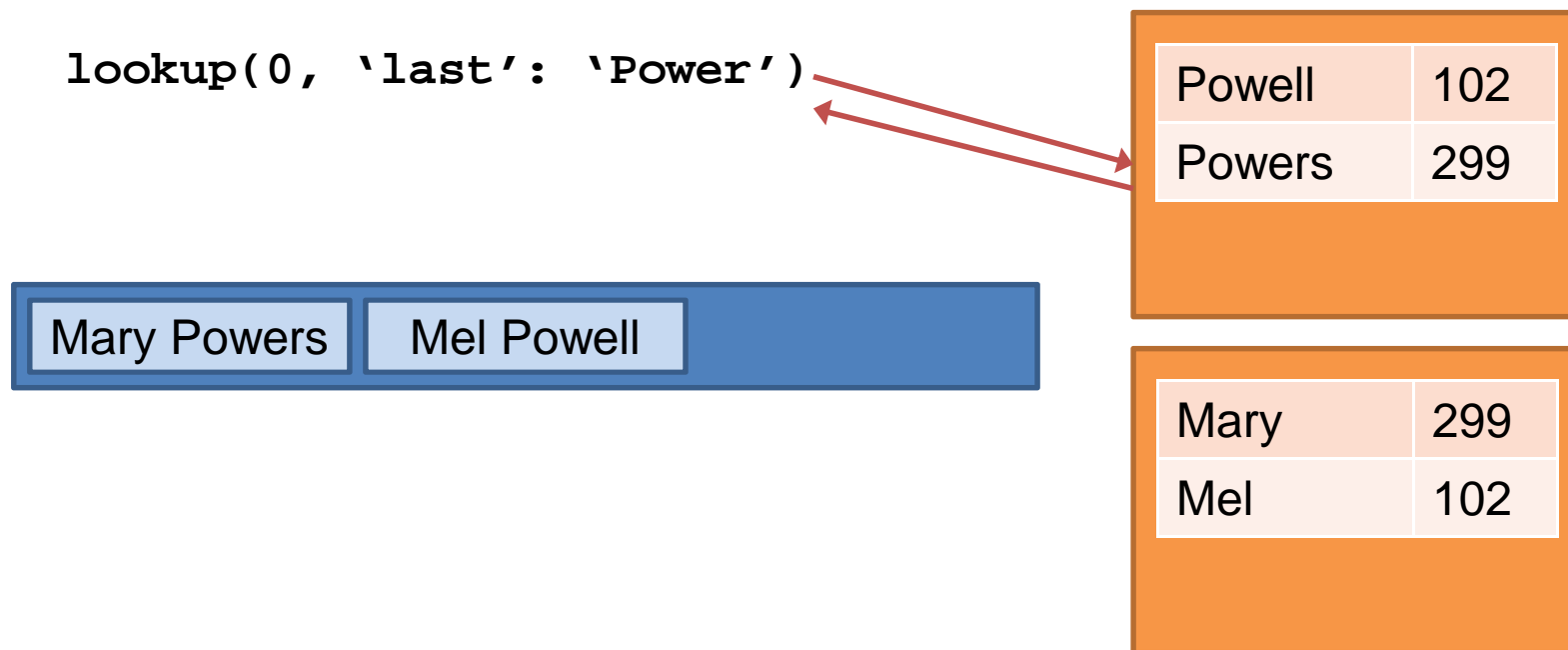
- **Our decision (for now): On search key**
 1. Don't want weakest-link lookup performance
 2. To support enumerate and cursors for range queries

Consistency

- **Problem: Index/Object inconsistency on puts**
 - Since 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 yet written
- **Avoid fancy commit protocol, if possible**
- **Idea: Index entries “commit” on object put**
 - Object puts are atomic
 - Index entries invalid until corresponding put finishes
- **Turns atomic puts into atomic index updates**

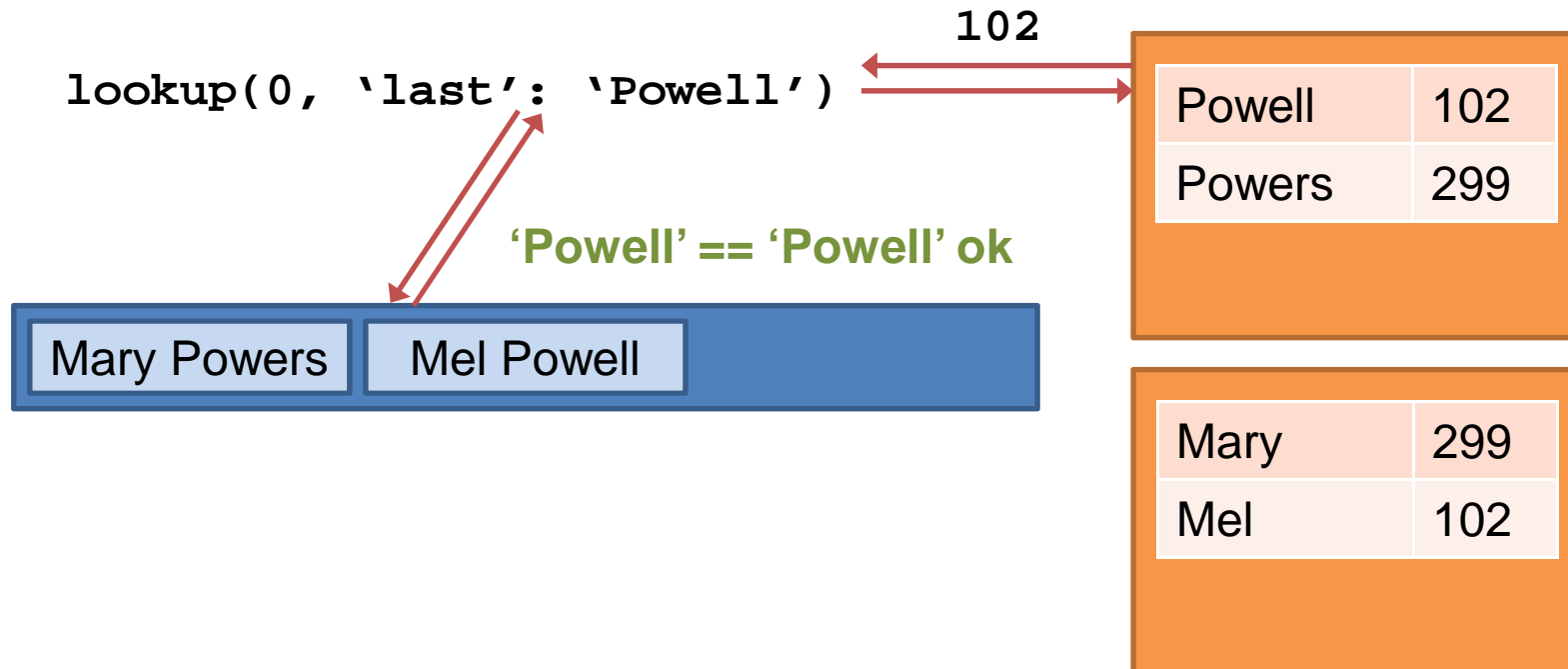
Consistency: Lookup

- **Request goes directly to correct index partition**
 - “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())
```

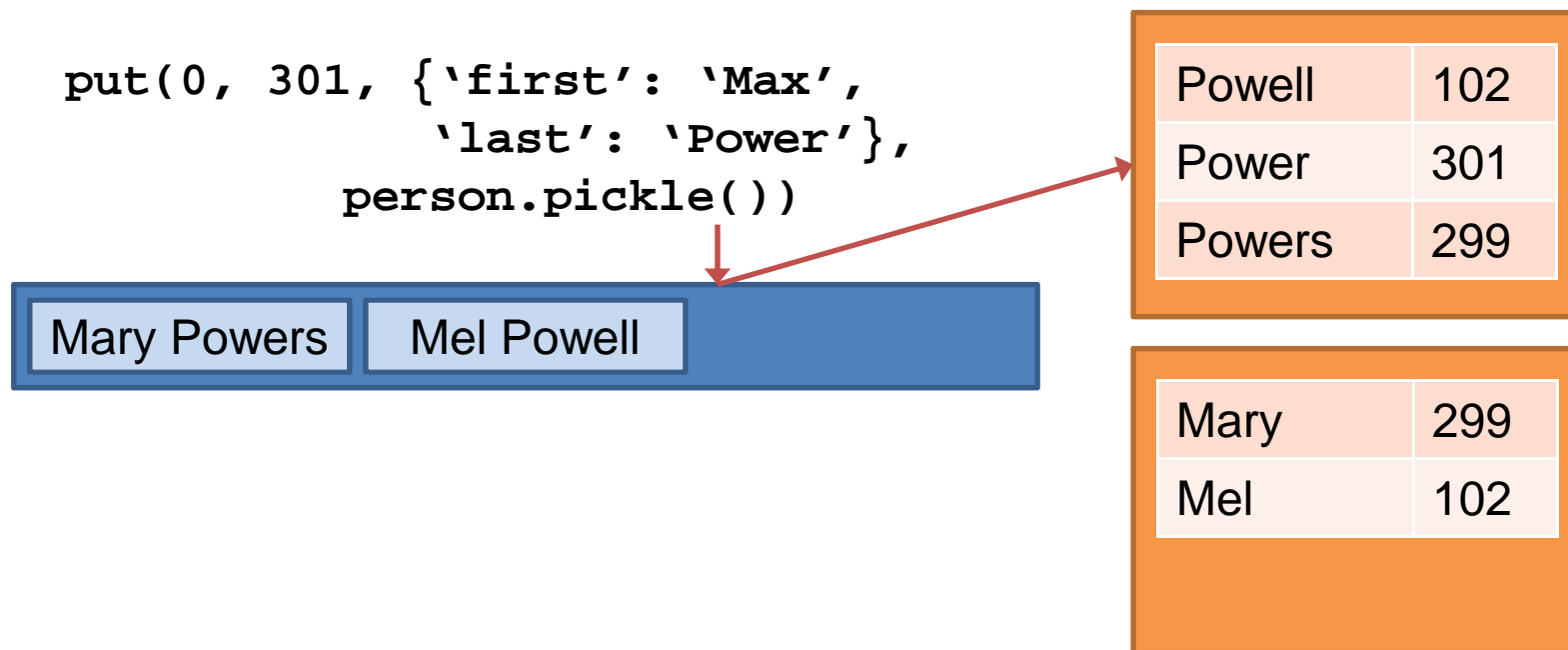


Powell	102
Powers	299

Mary	299
Mel	102

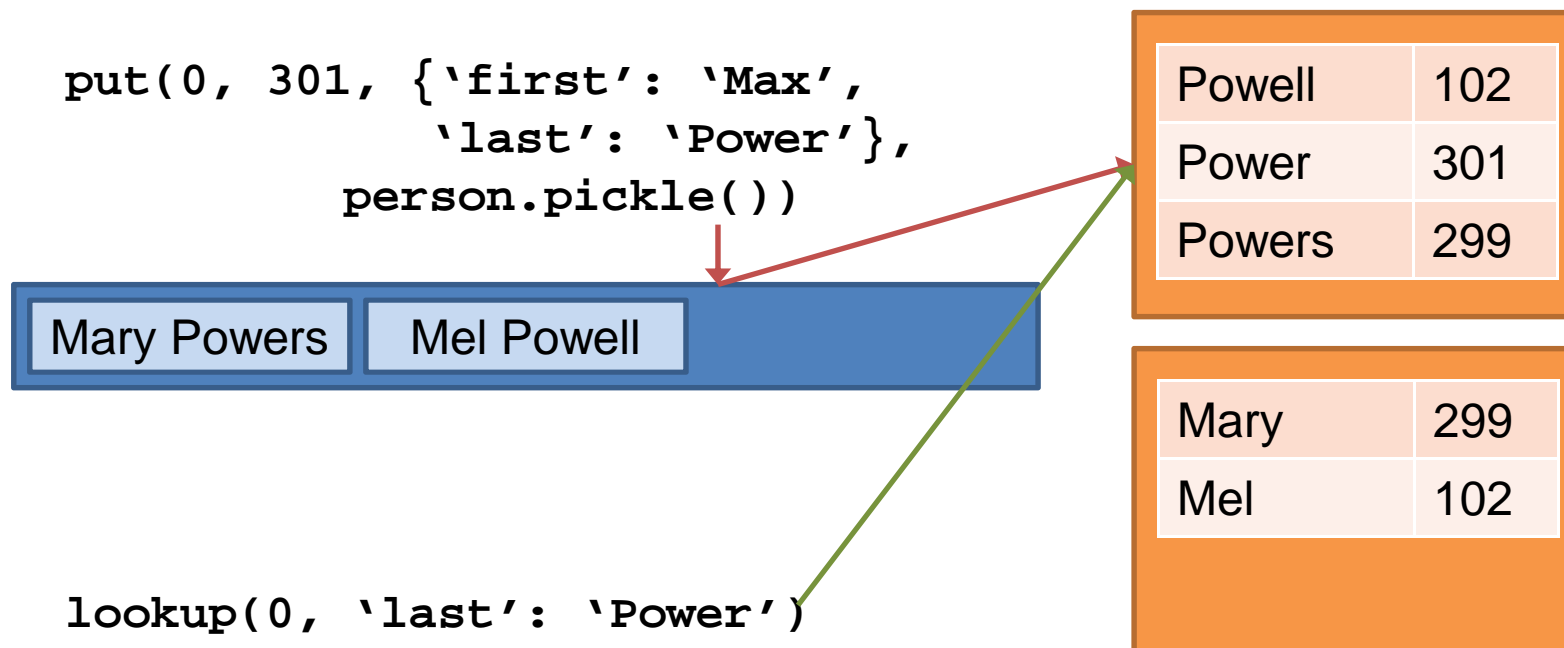
Consistency: Create

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



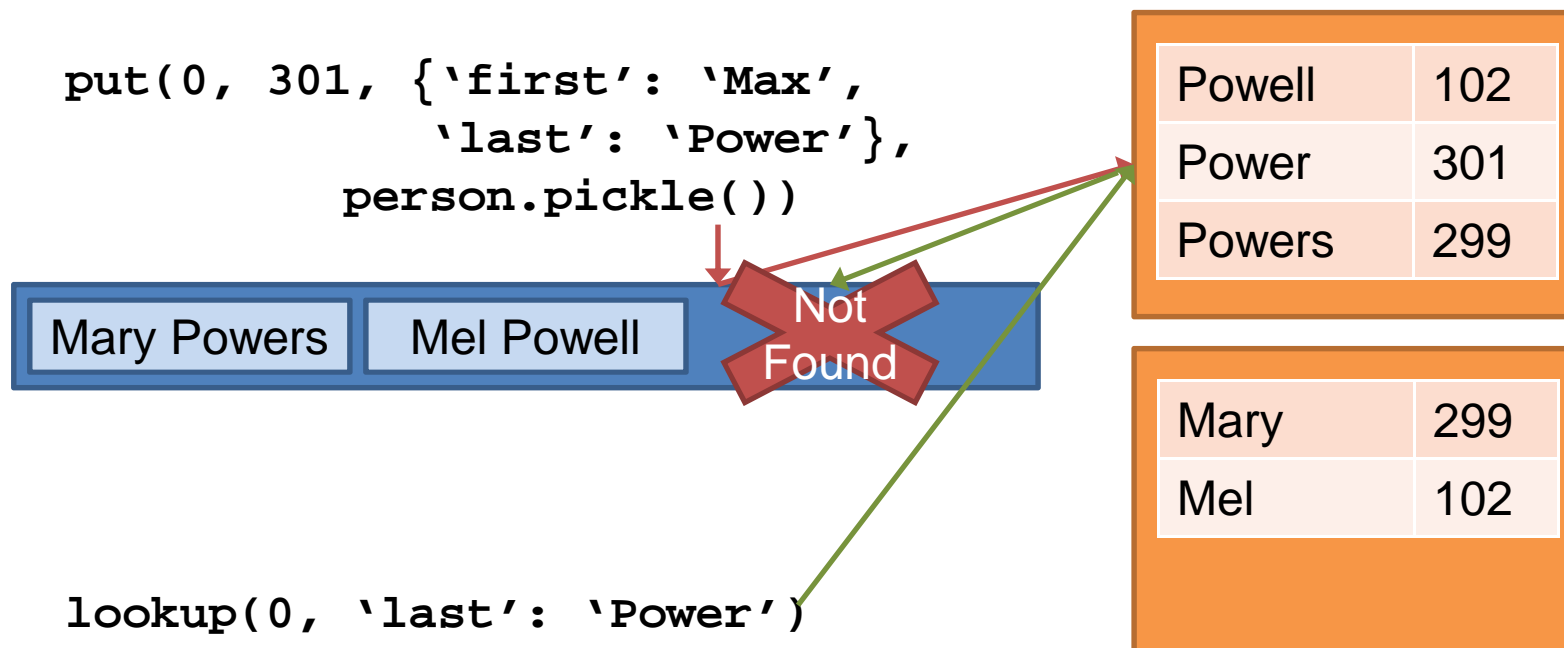
Consistency: Concurrent Lookup

- Concurrent ops ignore inconsistent entries



Consistency: Concurrent Lookup

- Concurrent ops ignore inconsistent entries



Consistency: Create (continued)

- Insert index entries before writing object

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



Powell	102
Power	301
Powers	299

Mary	299
Max	301
Mel	102

Consistency: Create

- Put completes; index entries now valid

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

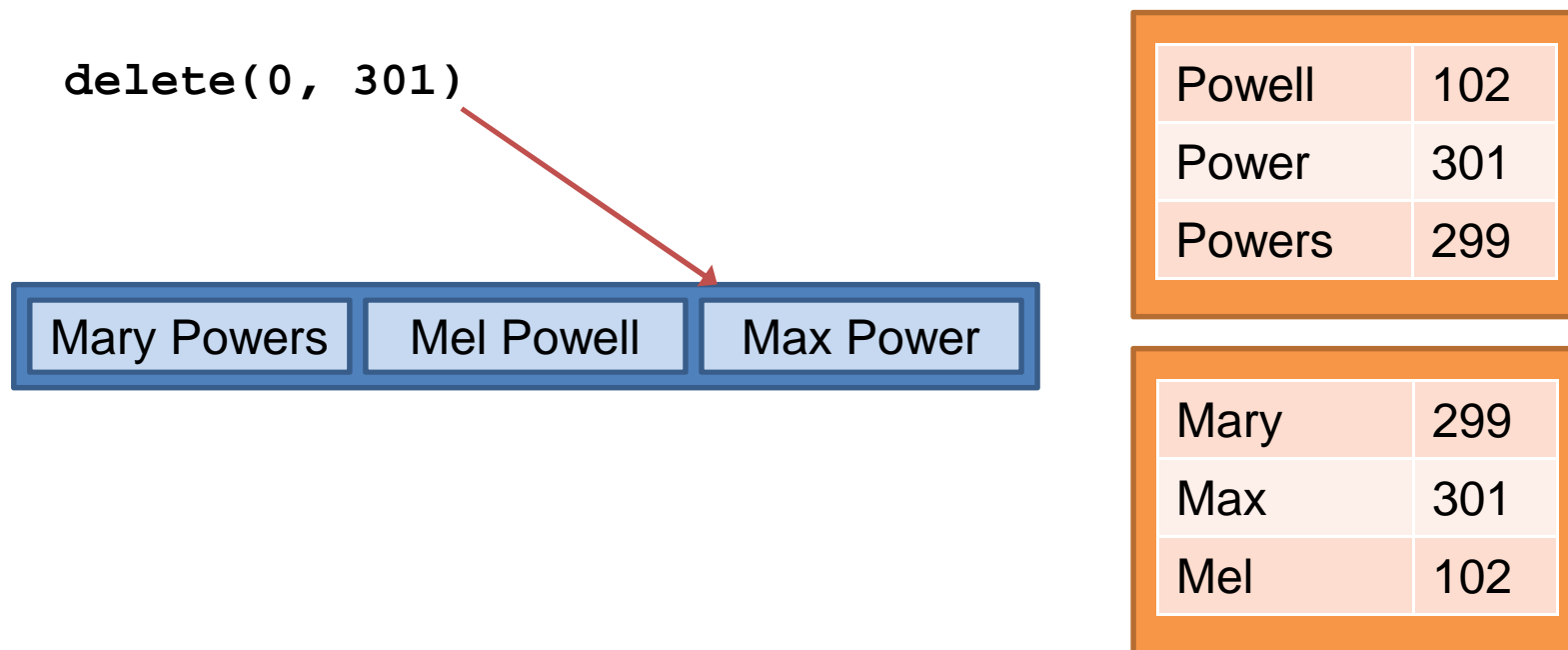


Powell	102
Power	301
Powers	299

Mary	299
Max	301
Mel	102

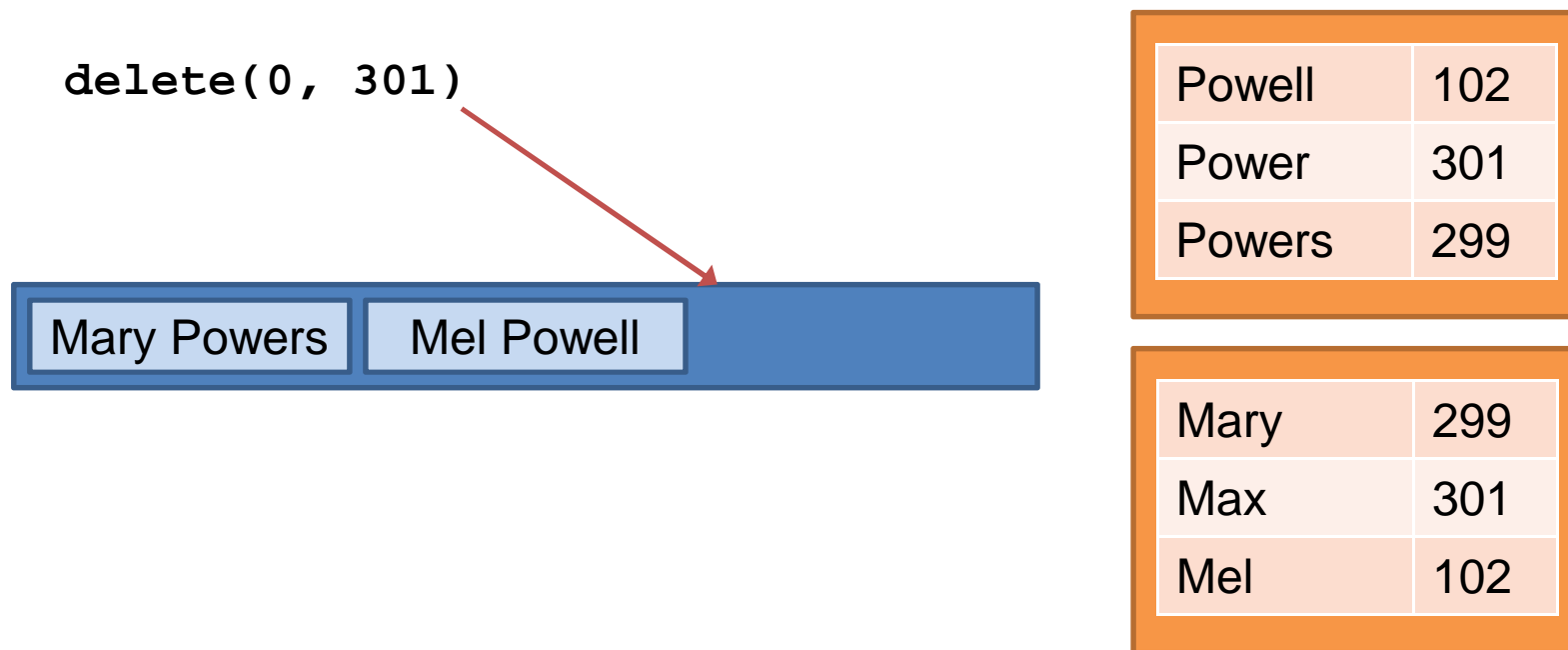
Consistency: Delete

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



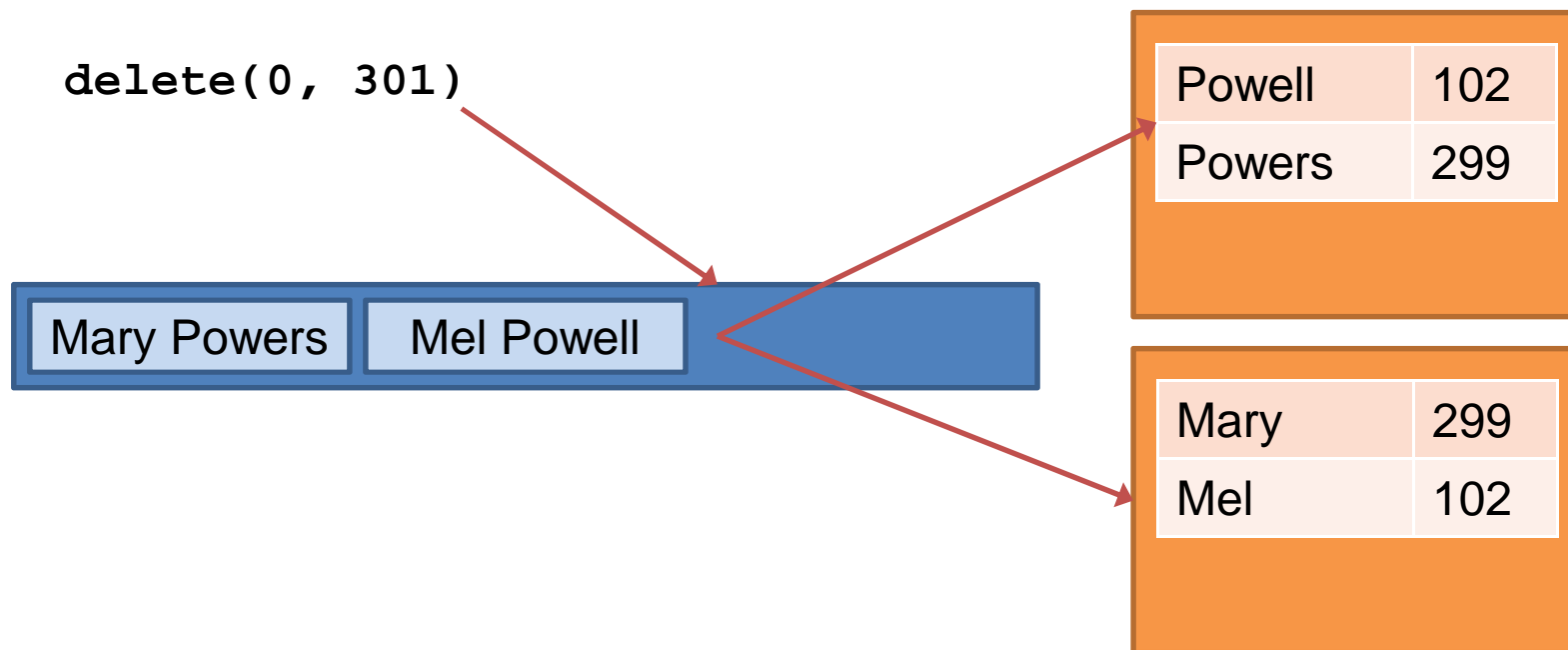
Consistency: Delete

- **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': 'Bowers'},  
     person.pickle())
```

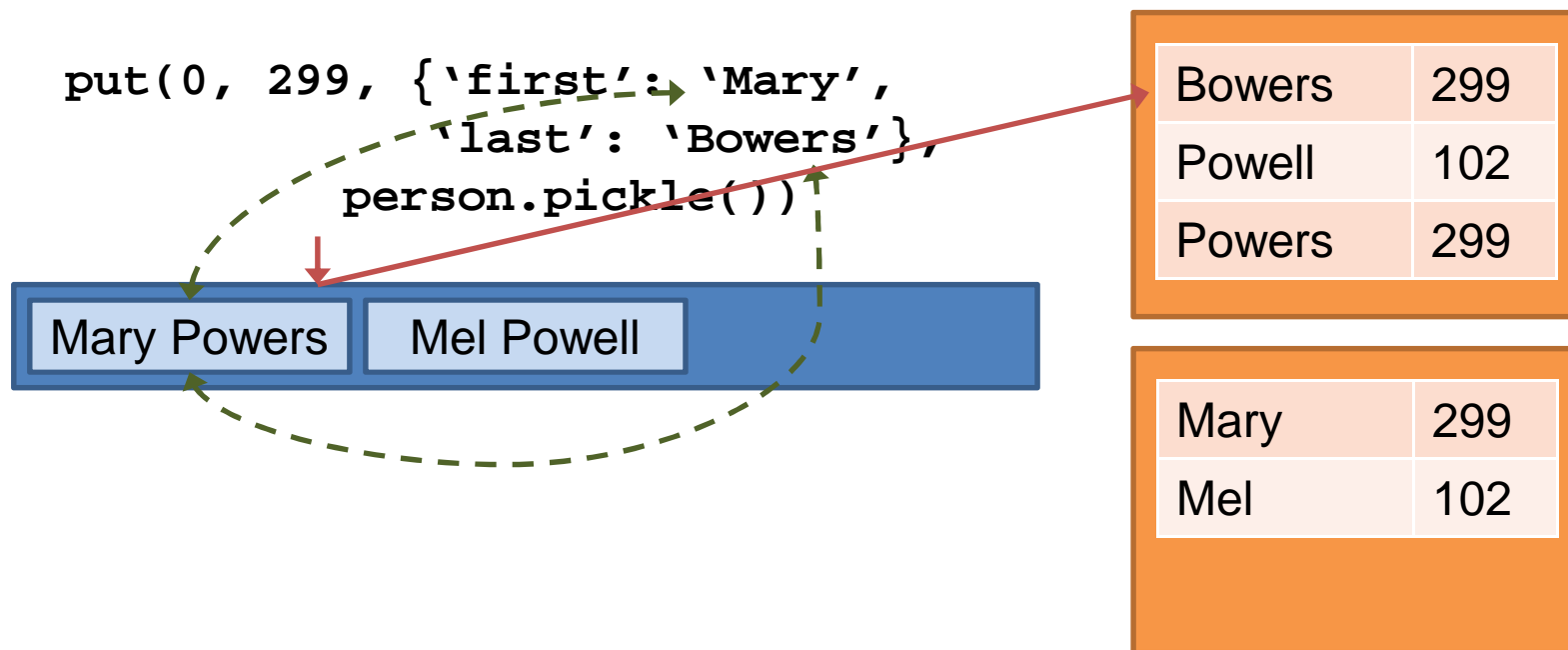


Powell	102
Powers	299

Mary	299
Mel	102

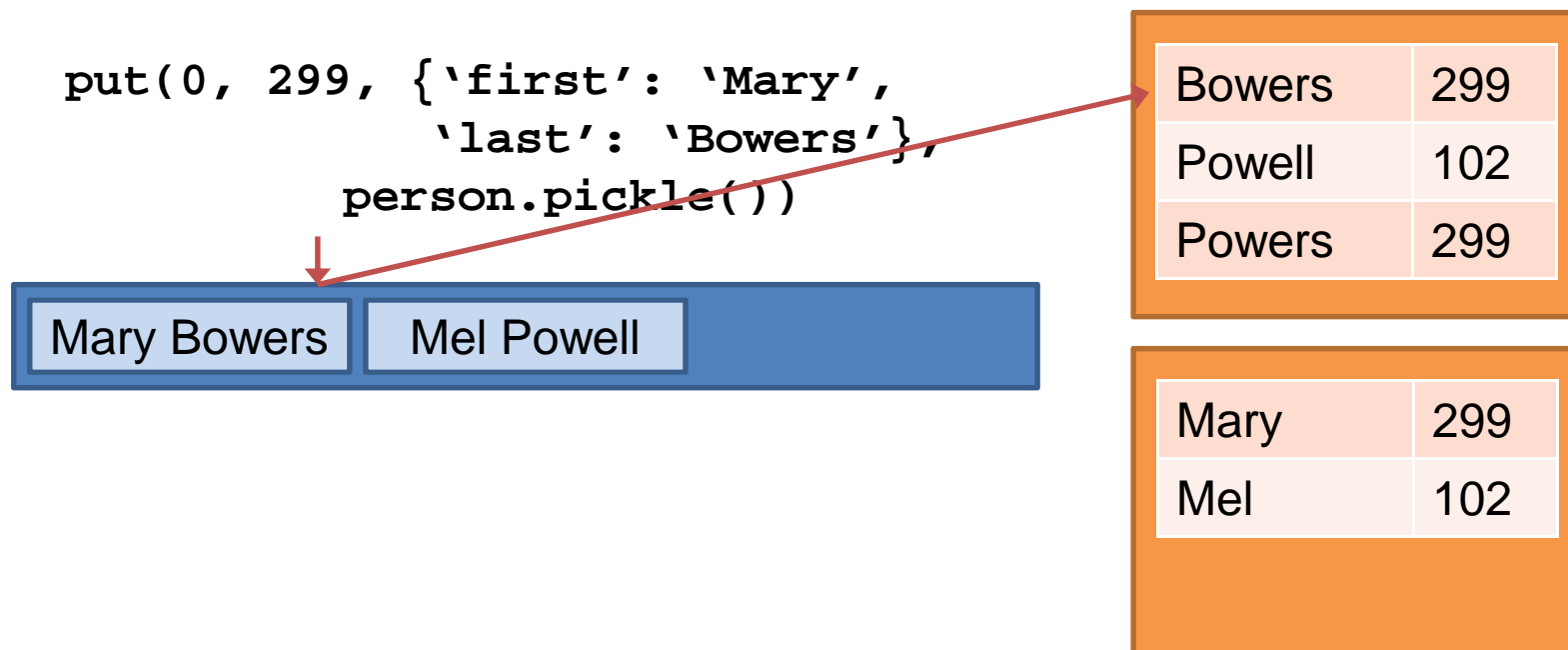
Consistency: Update

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



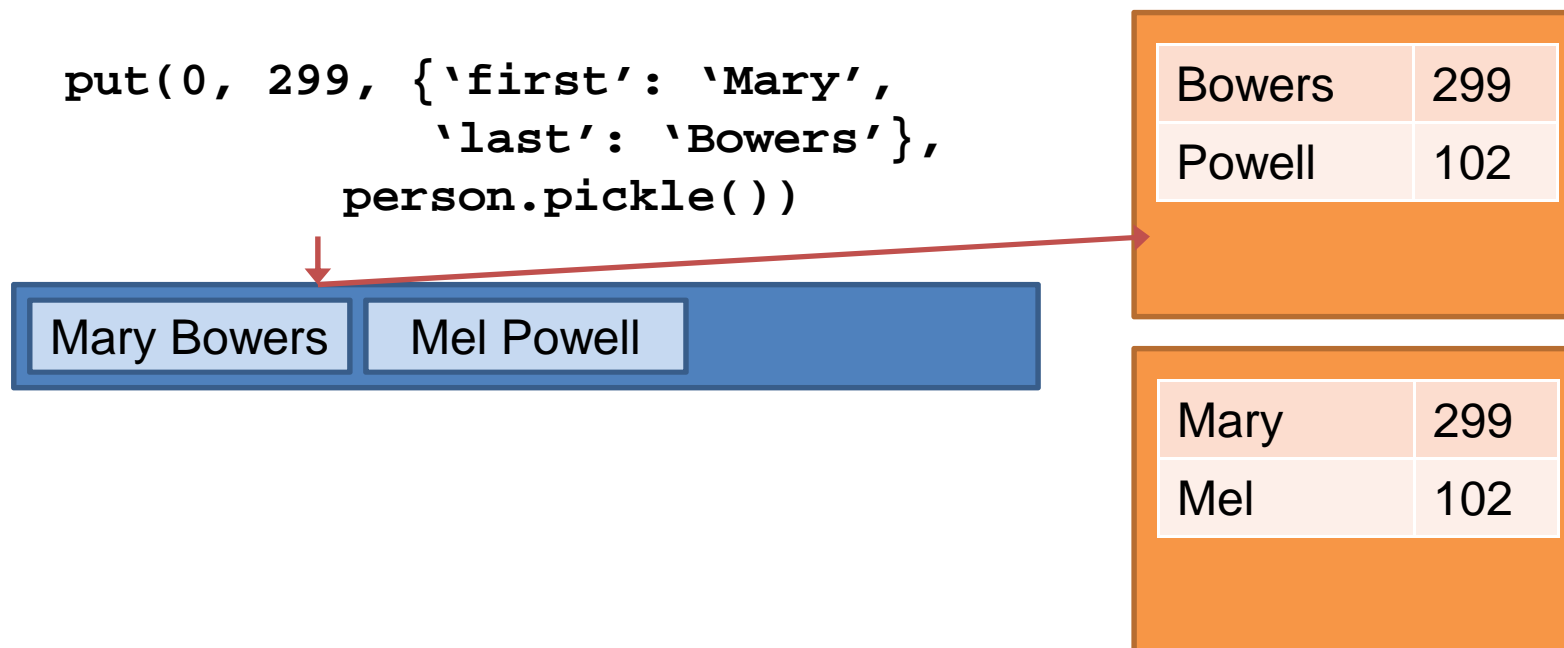
Consistency: Update

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



Consistency: Update

- Cleanup old, inconsistent entries



Consistency: Thoughts

- **Low-latency gives simplified consistency**
 - Can afford to have a single writer per object
- **Turn atomic puts into atomic index updates**
 - All index updates for an object go through master
 - Index entries invalid until corresponding put completes

THIS SLIDE IS REDUNDANT
BUT A NICE PLACE TO
TRANSITION TO NEXT TOPIC

Index Recovery

- **Problem: Unavailable until indexes recover**
 - Many requests will be lookups
 - These will block unless indexes are recovered
- **Rebuild versus Store?**
 - Prefer soft-state wherever possible
 - Possible using scale we can rebuild faster than store

Index Recovery: Sharding

- **How far does sharding + rebuilding get us?**
- **Assume indexes split on search key**
 - If split on object ID we do even better
- **Worst case: Entire shards of index data only**
 - At most 640 MB apiece

Index Recovery: Sharding Performance

Recover a single index shard on a new master:

- 1. Table shards recover elsewhere (0 – 0.6s)**
- 2. Table masters scan, extract index entries (0.6s)**
 - Hashtable: 10 million lookups/sec
 - 640 MB / 100 byte/object = 6,400,000 objects
 - $6,400,000 / 10,000,000 = 0.6 \text{ s}$
- 3. Transmit entries to new index master (0.6s)**
 - At most 640 MB
- 4. New index master rebuilds index (0.6s)**
 - Similar time to master hashtable scan
- **At least 2, 3, & 4 can happen in parallel**
 - 0.6s to recover table + 0.6s to scan, extract, transmit, rebuild
 - **1.2s upper bound for conservative 100b object size**

Summary

- **Explicit search keys both flexible and efficient**
- **Partition indexes on search key for easy lookup/enumeration**
- **Atomic puts simplify atomic indexes**
- **Scale drives fast recovery for availability**