Amazon DynamoDB

Sean Shriver
NoSQL Solutions Architect
Amazon Web Services

September 2016

Agenda

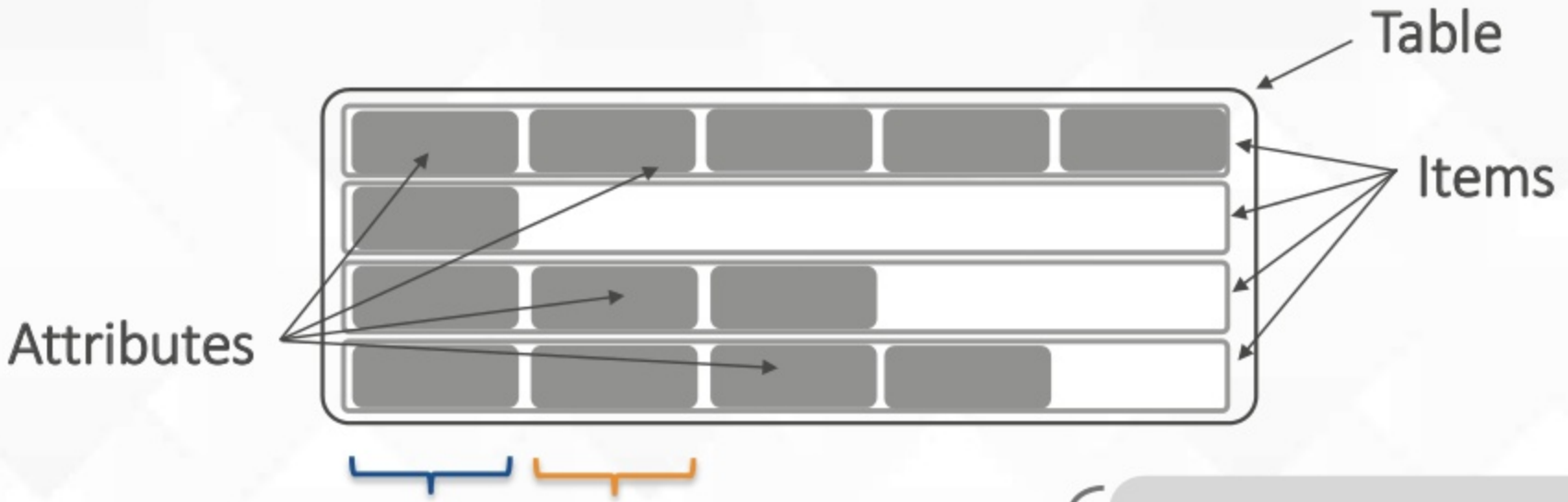
- Tables, API, data types, indexes
- Scaling
- Data modeling
- Scenarios and best practices
- DynamoDB Streams
- Reference architecture

Amazon DynamoDB

- Managed NoSQL database service
- Supports both document and key-value data models
- Highly scalable
- Consistent, single-digit millisecond latency at any scale
- Highly available—3x replication
- Simple and powerful API

Tables, Partitioning

Table



Partition Key

Mandatory
Key-value access pattern
Determines data distribution

Sort Key

Optional
Model 1:N relationships
Enables rich query capabilities

All items for a partition key
==, <, >, >=, <=
“begins with”
“between”
sorted results
counts
top/bottom N values
paged responses

Table and item API

- CreateTable
- UpdateTable
- DeleteTable
- DescribeTable
- ListTables

- GetItem
- Query
- Scan
- BatchGetItem

- PutItem
- UpdateItem
- DeleteItem
- BatchWriteItem

DynamoDB

Stream API

- ListStreams
- DescribeStream
- GetShardIterator
- GetRecords

Data types

- String (S)
- Number (N)
- Binary (B)

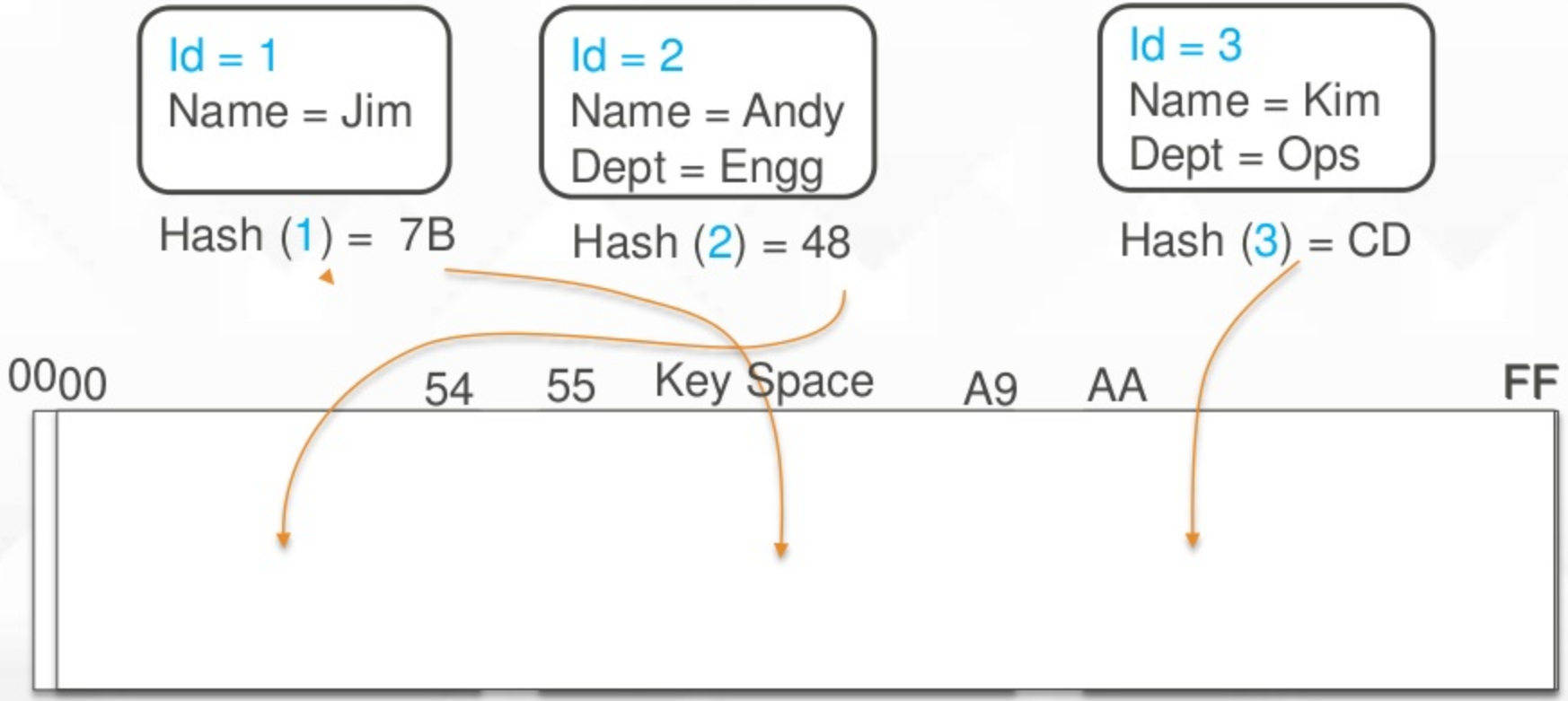
- String Set (SS)
- Number Set (NS)
- Binary Set (BS)

- Boolean (BOOL)
- Null (NULL)
- List (L)
- Map (M)

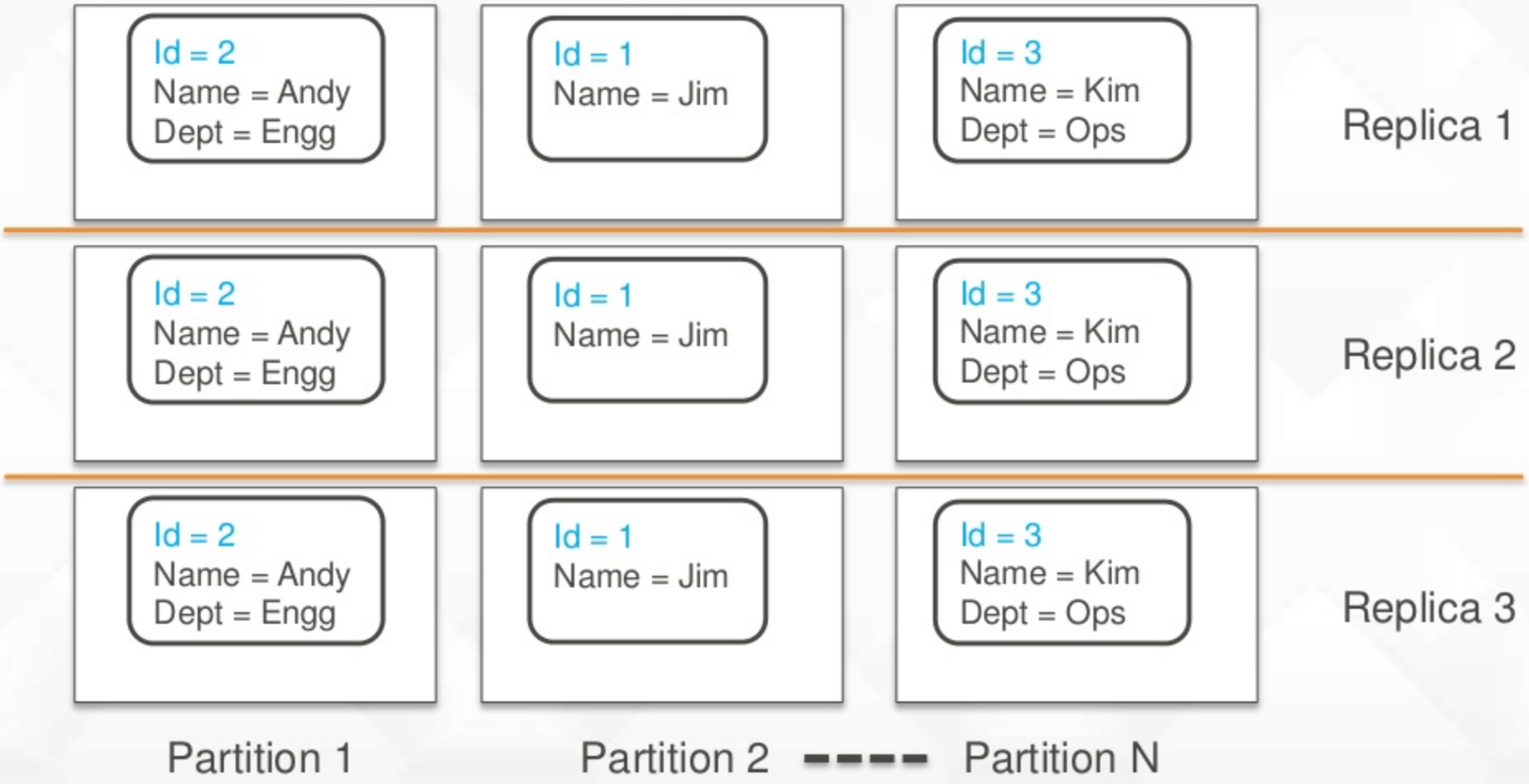
Used for storing nested JSON documents

Partition table

- Partition key uniquely identifies an item
- Partition key is used for building an unordered hash index
- Table can be partitioned for scale

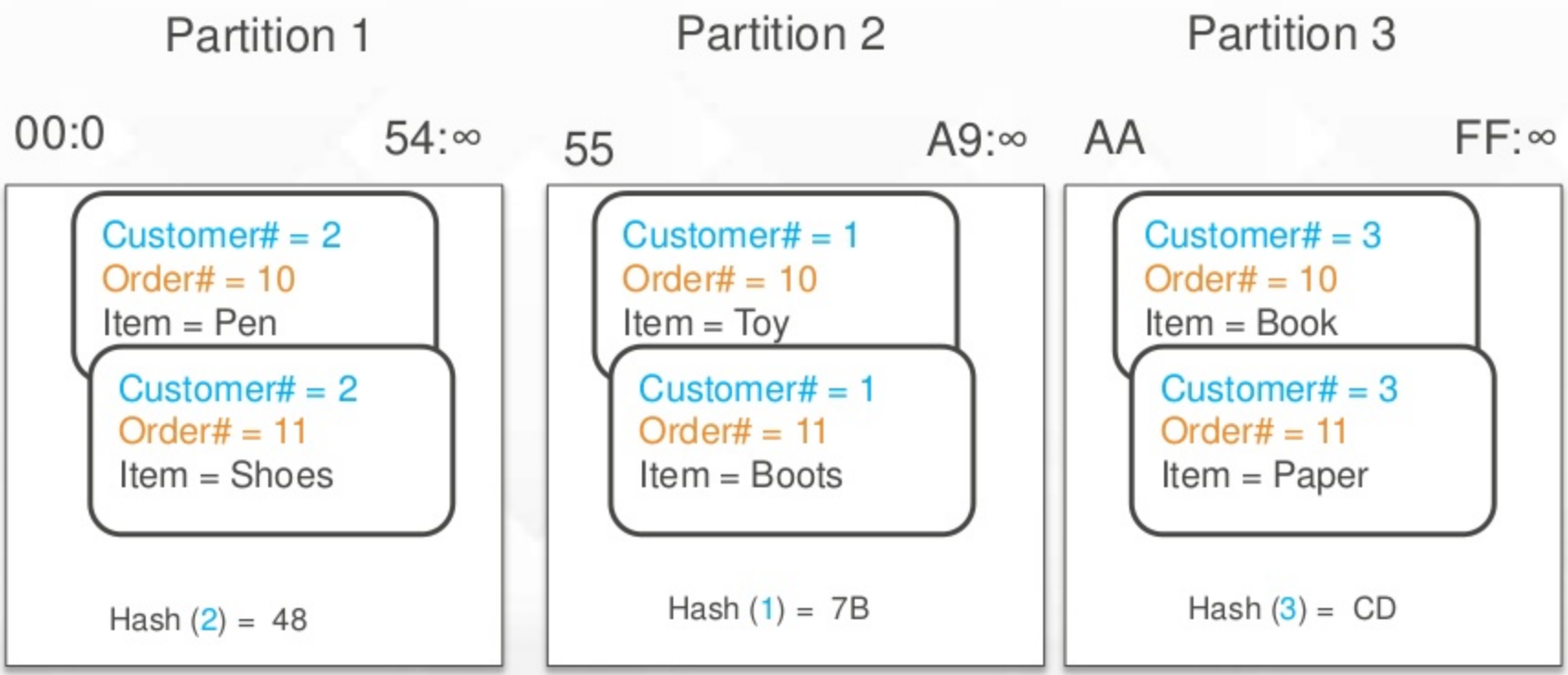


Partitions are three-way replicated



Partition-sort key table

- Partition key and sort key together uniquely identify an Item
- Within unordered partition key-space, data is sorted by the sort key
- No limit on the number of items (∞) per partition key
  - Except if you have local secondary indexes

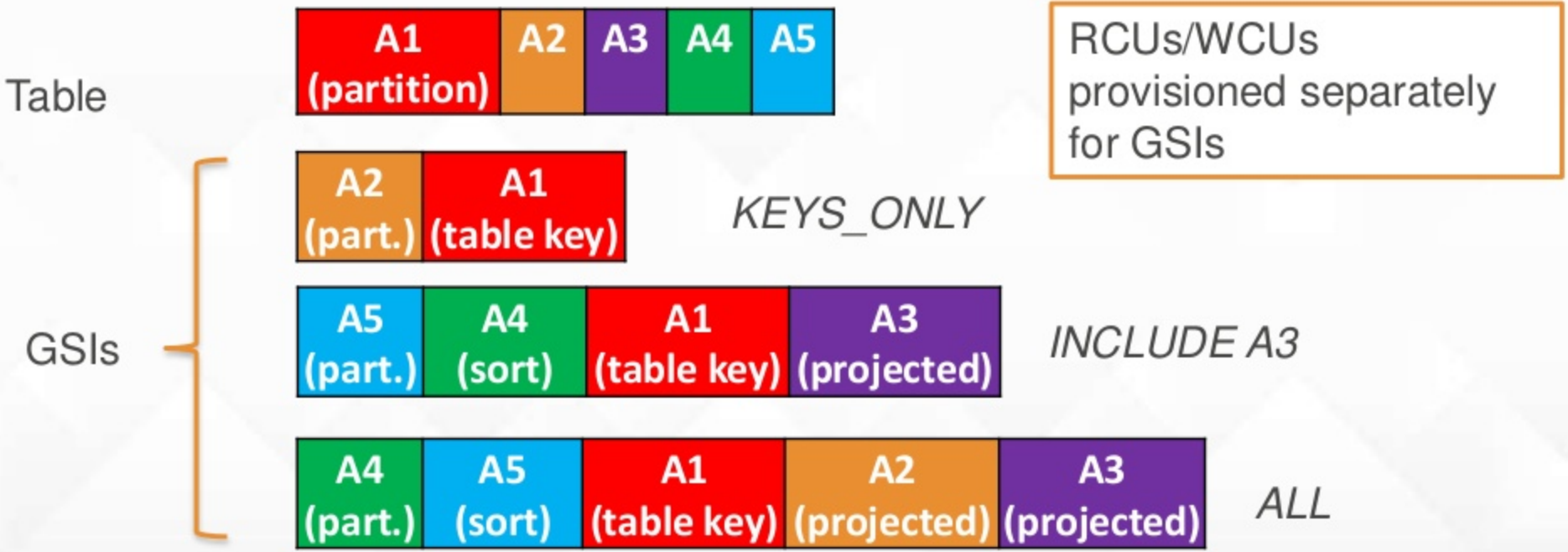


Indexes

Global secondary index (GSI)

Online Indexing

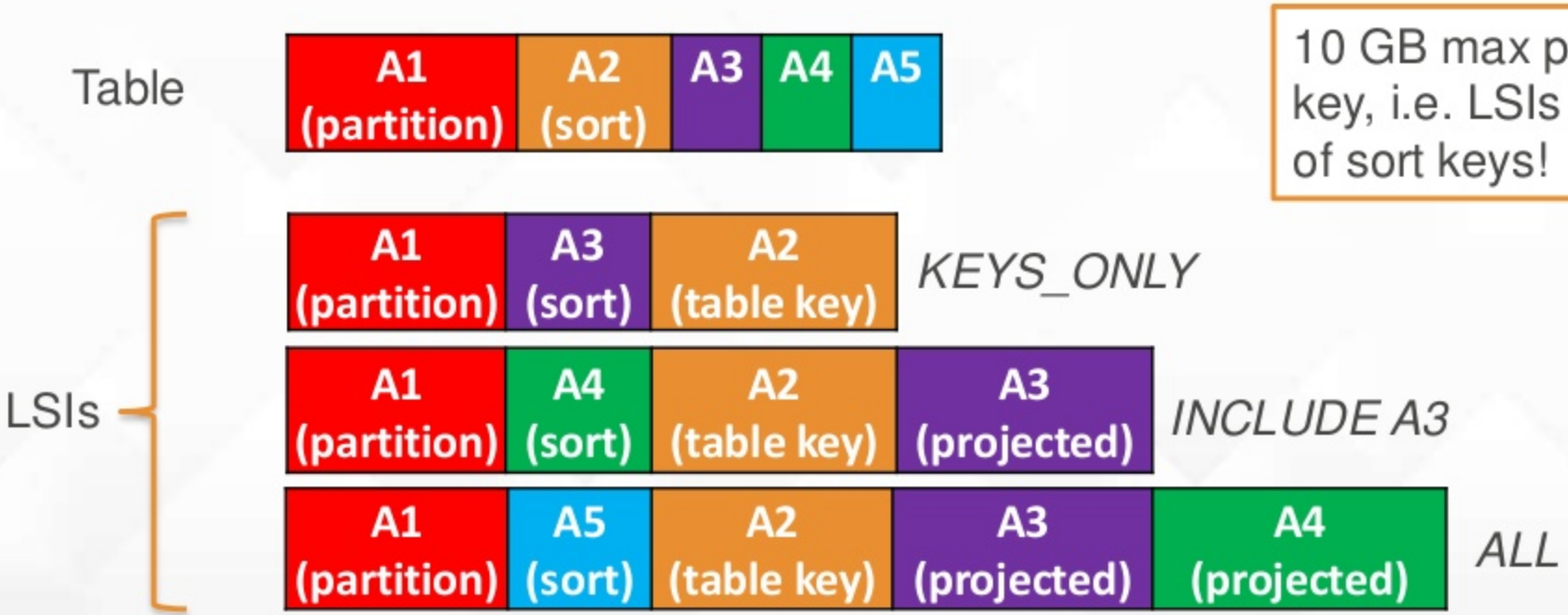
- Alternate partition (+sort) key
- Index is across all table partition keys



RCUs/WCUs provisioned separately for GSIs

Local secondary index (LSI)

- Alternate sort key attribute
- Index is local to a partition key



Scaling

Scaling

- Throughput
  - Provision any amount of throughput to a table
- Size
  - Add any number of items to a table
    - Max item size is 400 KB
    - LSIs limit the number of items due to 10 GB limit
- Scaling is achieved through partitioning

Throughput

- Provisioned at the table level
  - Write capacity units (WCUs) are measured in 1 KB per second
  - Read capacity units (RCUs) are measured in 4 KB per second
    - RCUs measure strictly consistent reads
    - Eventually consistent reads cost 1/2 of consistent reads
- Read and write throughput limits are independent

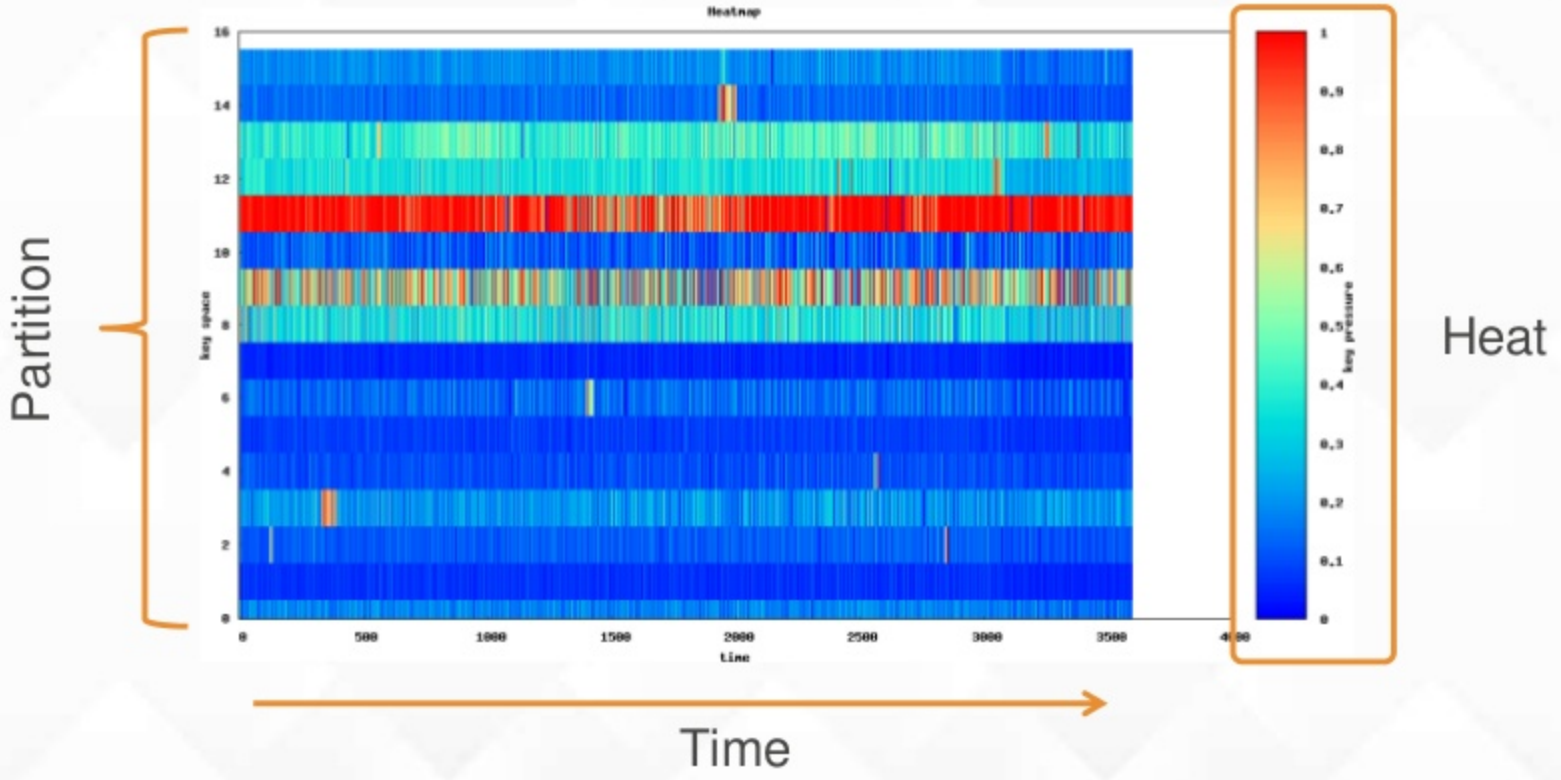
Getting the most out of DynamoDB throughput

"To get the most out of DynamoDB throughput, create tables where the partition key has a large number of distinct values, and values are requested fairly uniformly, as randomly as possible."

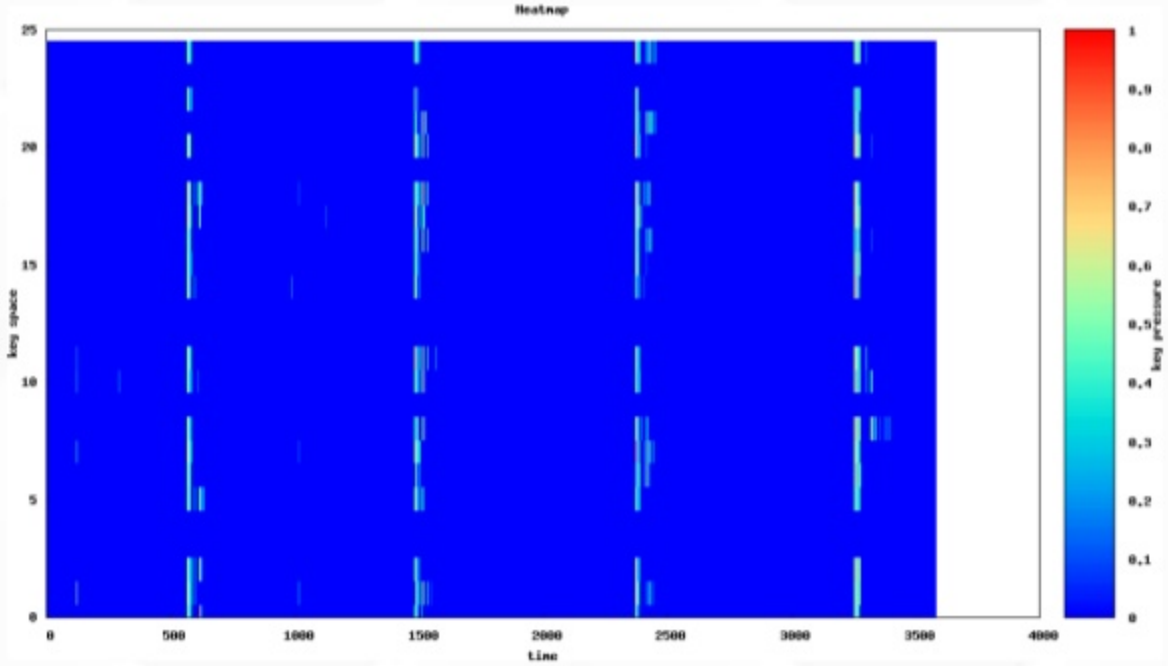
—*DynamoDB Developer Guide*

1. Key Choice: High key cardinality
2. Uniform Access: access is evenly spread over the key-space
3. Time: requests arrive evenly spaced in time

Example: Key Choice or Uniform Access

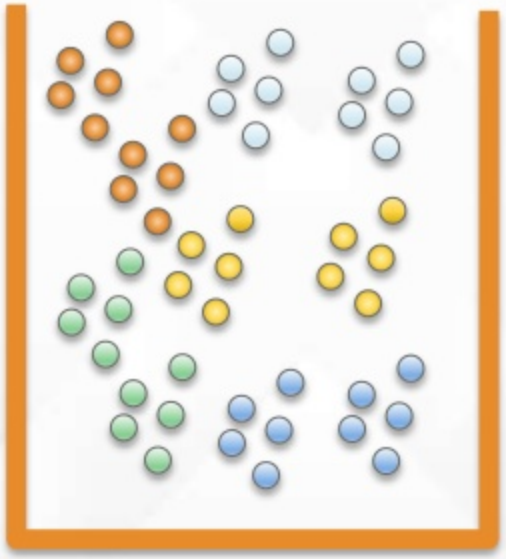


Example: Time



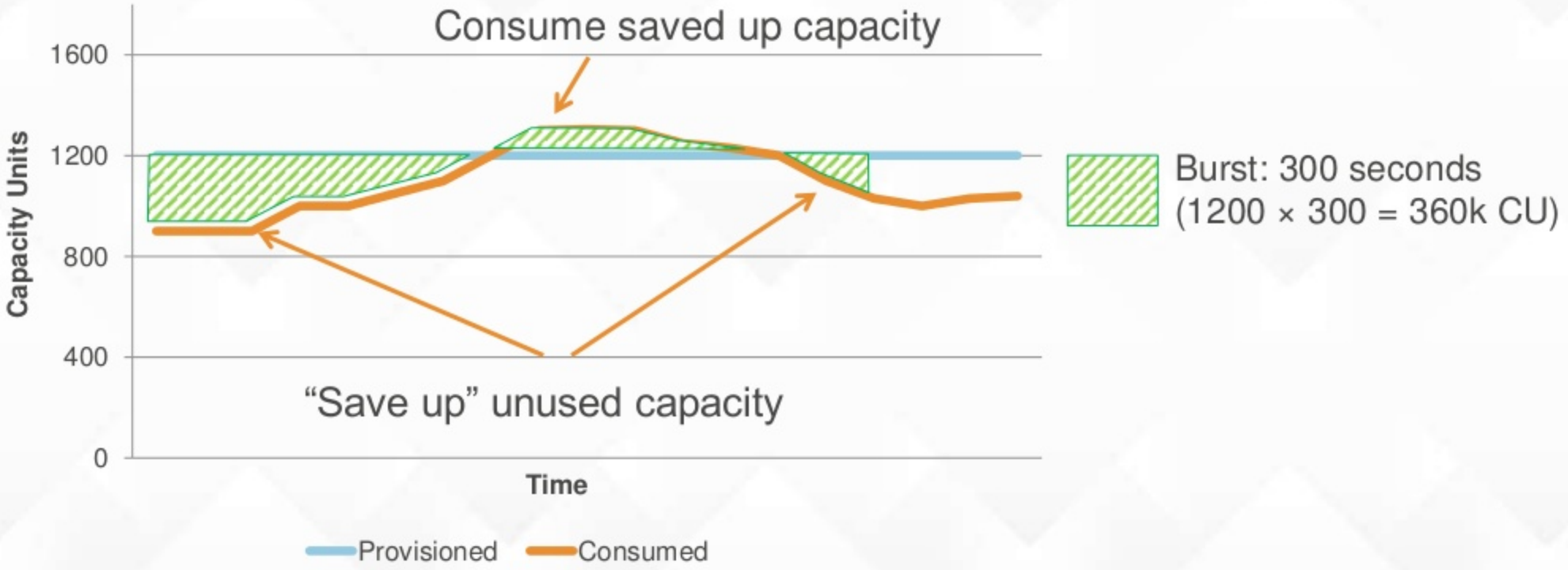
How does DynamoDB handle bursts?

- DynamoDB saves 300 seconds of *unused* capacity per partition



Bursting is best effort!

Burst capacity is built-in



Burst capacity may not be sufficient

Don’t completely depend on burst capacity… provision sufficient throughput

What causes throttling?

- If **sustained** throughput goes beyond provisioned throughput per partition

- From the example before:
  - Table created with 5000 RCUs, 500 WCUs
  - RCUs per partition = 1666.67
  - WCUs per partition = 166.67
  - If sustained throughput > (1666 RCUs or 166 WCUs) per key or partition, DynamoDB may throttle requests
    - Solution: Increase provisioned throughput

What causes throttling?

- Non-uniform workloads
  - Hot keys/hot partitions
  - Very large bursts
- Dilution of throughout across partitions caused by mixing hot data with cold data
  - Use a table per time period for storing time series data so WCUs and RCUs are applied to the hot data set

Data Modeling

Store data based on how you will access it!

1:1 relationships or key-values

- Use a table or GSI with a partition key
- Use GetItem or BatchGetItem API

Example: Given a user or email, get attributes

Users Table	
Partition key	Attributes
UserId = bob	Email = bob@gmail.com, JoinDate = 2011-11-15
UserId = fred	Email = fred@yahoo.com, JoinDate = 2011-12-01

Users-Email-GSI	
Partition key	Attributes
Email = bob@gmail.com	UserId = bob, JoinDate = 2011-11-15
Email = fred@yahoo.com	UserId = fred, JoinDate = 2011-12-01

1:N relationships or parent-children

- Use a table or GSI with partition and sort key
- Use Query API

Example: Given a device, find all readings between epoch X, Y

Device-measurements		
Part. Key	Sort key	Attributes
DeviceId = 1	epoch = 5513A97C	Temperature = 30, pressure = 90
DeviceId = 1	epoch = 5513A9DB	Temperature = 30, pressure = 90

N:M relationships

- Use a table and GSI with partition and sort key elements switched
- Use Query API

Example: Given a user, find all games. Or given a game, find all users.

User-Games-Table	
Part. Key	Sort key
UserId = bob	GameId = Game1
UserId = fred	GameId = Game2
UserId = bob	GameId = Game3

Game-Users-GSI	
Part. Key	Sort key
GameId = Game1	UserId = bob
GameId = Game2	UserId = fred
GameId = Game3	UserId = bob

Documents (JSON)

- Data types (M, L, BOOL, NULL) introduced to support JSON
- Document SDKs
  - Simple programming model
  - Conversion to/from JSON
  - Java, JavaScript, Ruby, .NET
- Cannot create an Index on elements of a JSON object stored in Map
  - They need to be modeled as top-level table attributes to be used in LSIs and GSIs
- Set, Map, and List have no element limit but depth is 32 levels

Javascript	DynamoDB
string	S
number	N
boolean	BOOL
null	NULL
array	L
object	M

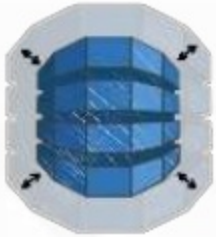
```
var image = { // JSON Object for an image
    imageid: 12345,
    url: 'http://example.com/awesome_image.jpg'
};
var params = {
    TableName: 'images',
    Item: image, // JSON Object to store
};
dynamodb.putItem(params, function(err, data){
    // response handler
};
```

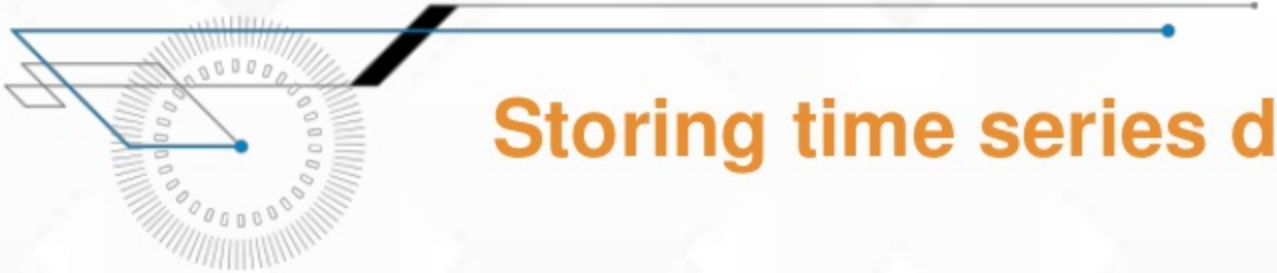
Rich expressions

- Projection expression
  - Query/Get/Scan: ProductReviews.FiveStar[0]
- Filter expression
  - Query/Scan: #V > :num (#V is a place holder for keyword VIEWS)
- Conditional expression
  - Put/Update/DeleteItem: attribute_not_exists (#pr.FiveStar)
- Update expression
  - UpdateItem: set Replies = Replies + :num

Scenarios and Best Practices

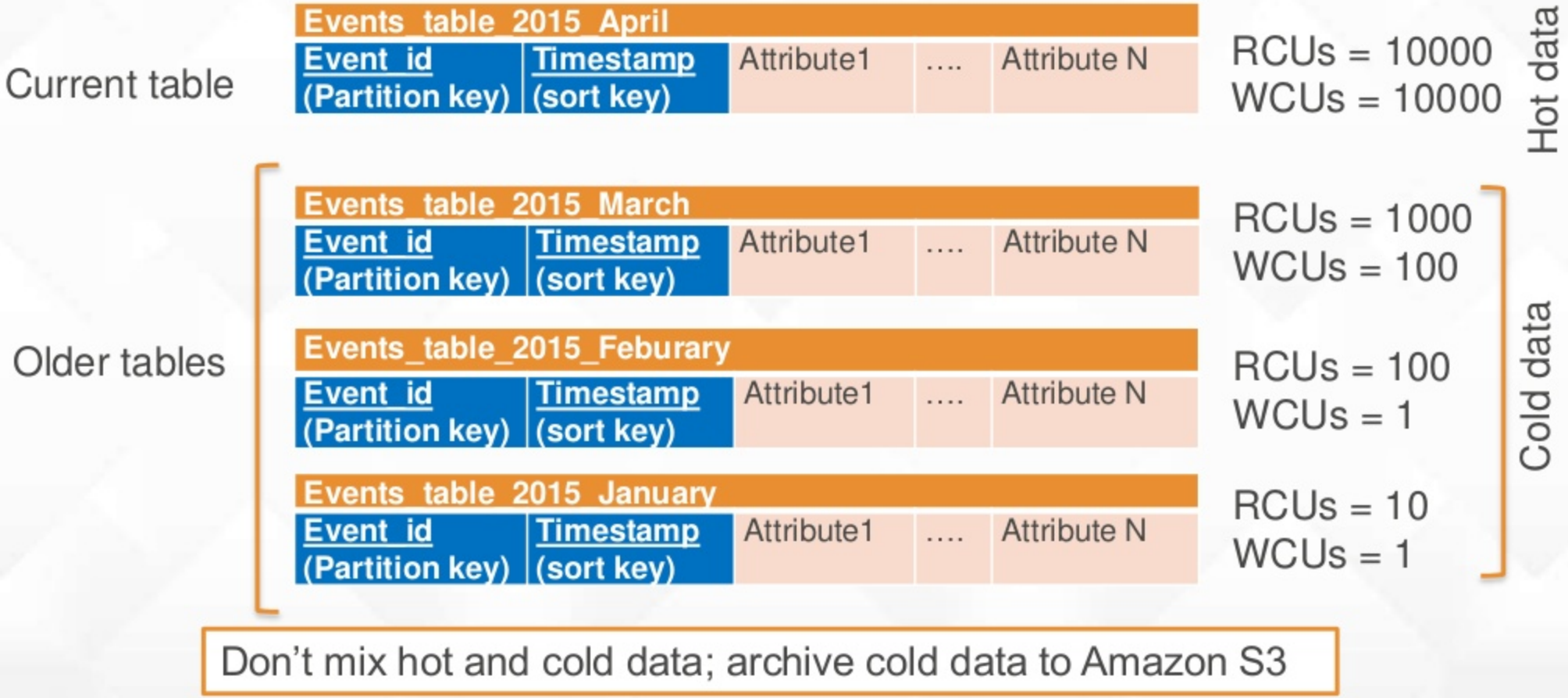
Event Logging





Storing time series data

Time series tables

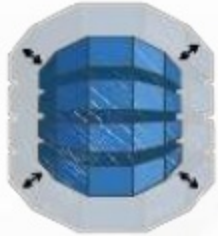


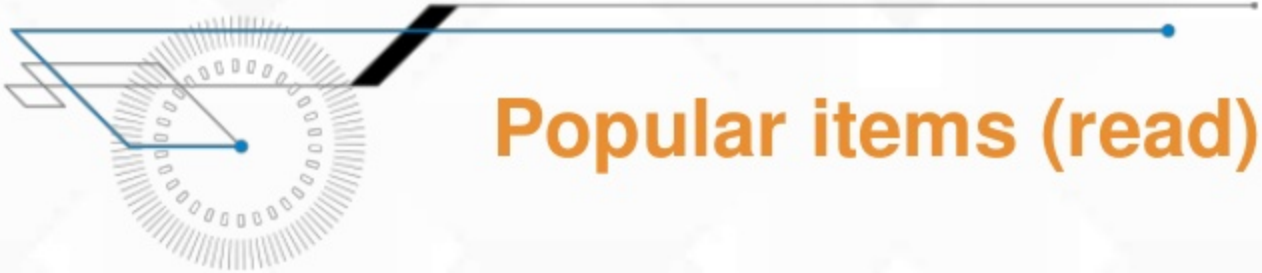
Use a table per time period

- Pre-create daily, weekly, monthly tables
- Provision required throughput for current table
- Writes go to the current table
- Turn off (or reduce) throughput for older tables

Important when: Dealing with time series data

Product Catalog



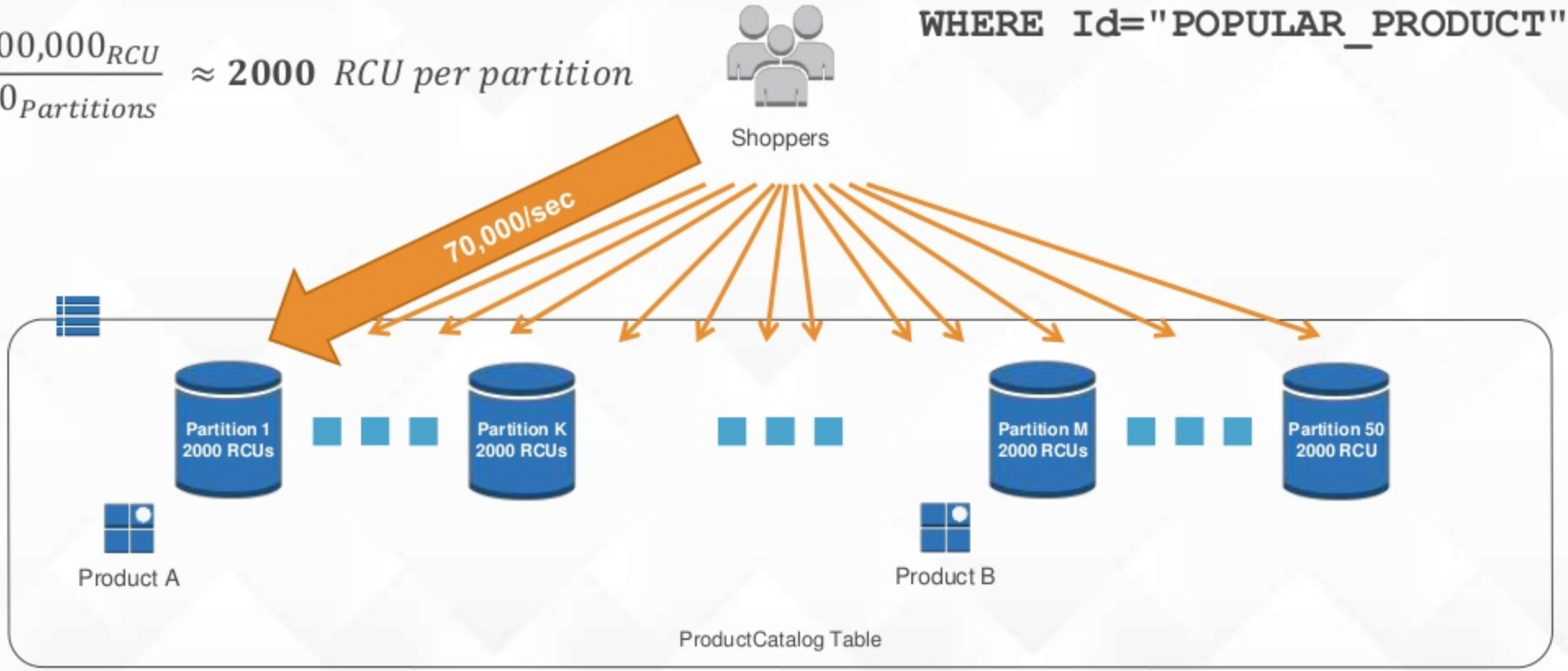


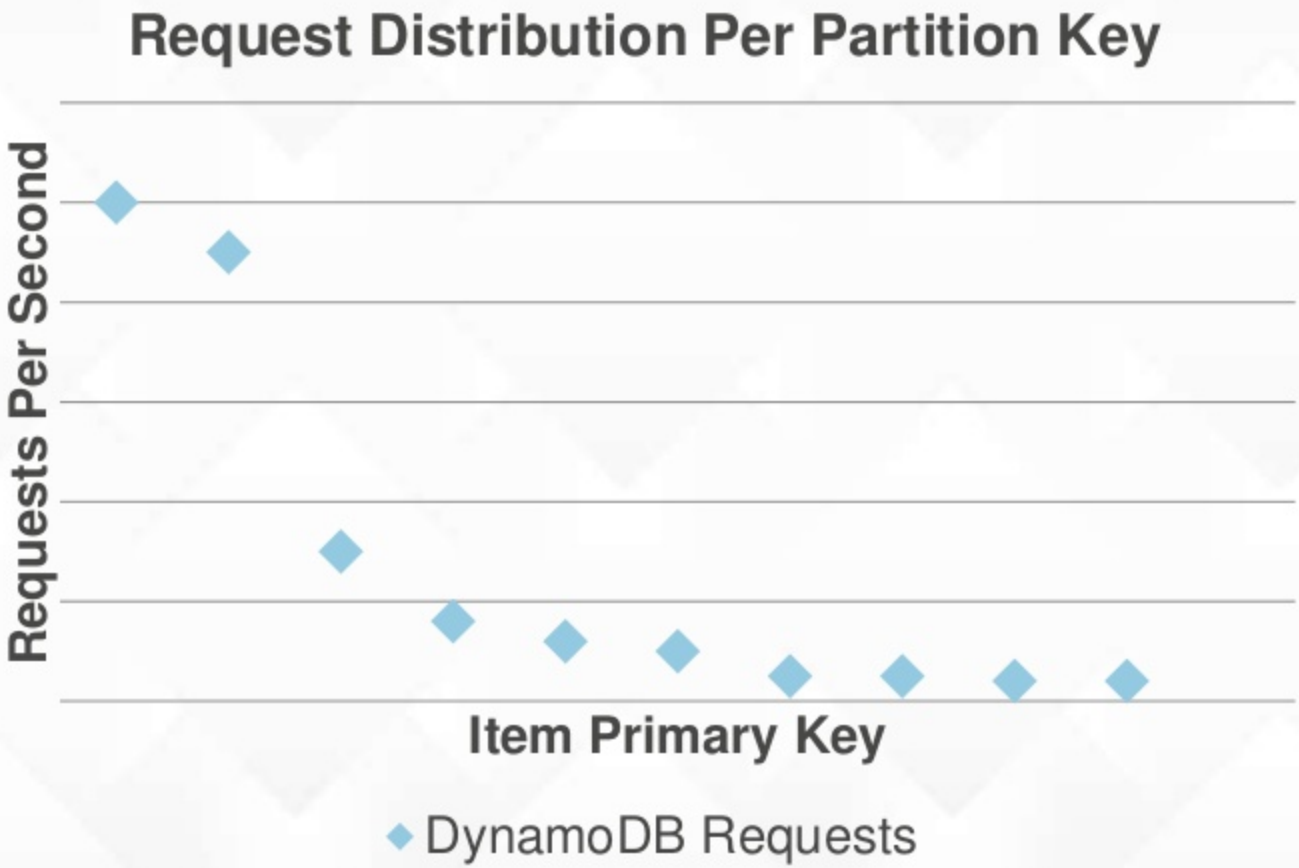
Popular items (read)

Scaling bottlenecks

$$\frac{100{,}000_{RCU}}{50_{Partitions}} \approx \mathbf{2000}\ RCU\ per\ partition$$

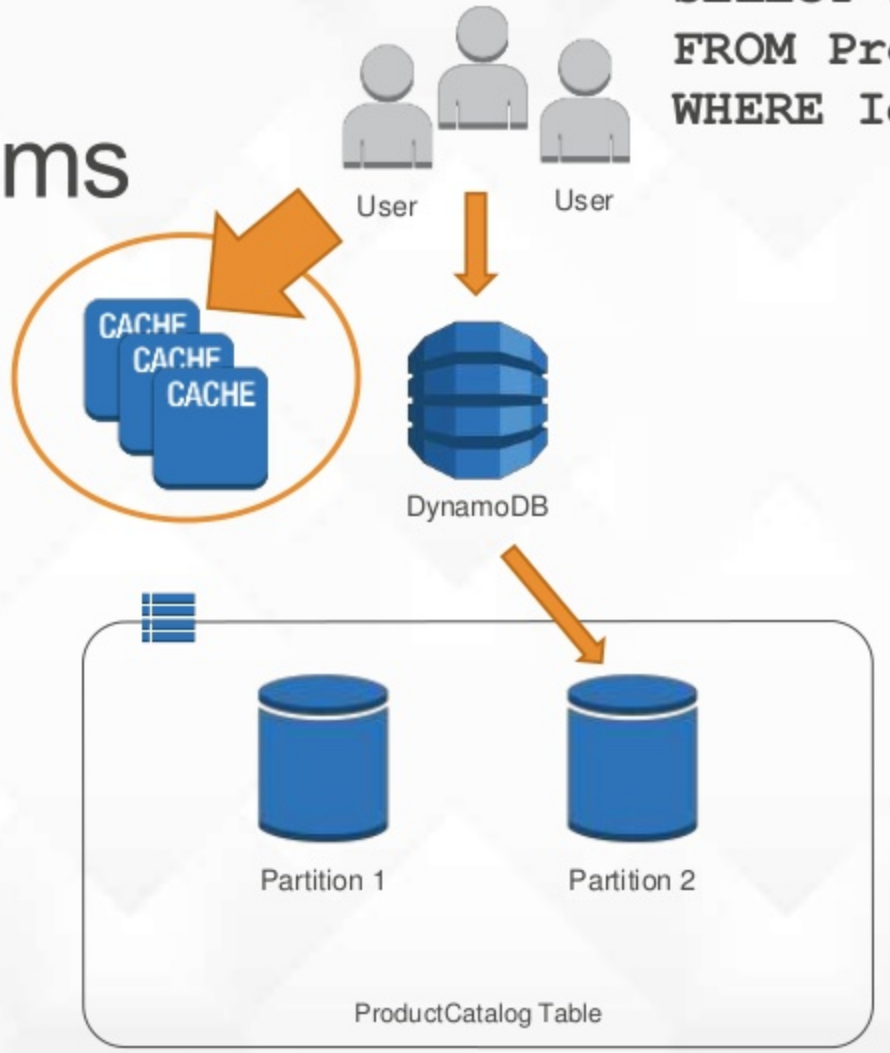
```
SELECT Id, Description, ...
FROM ProductCatalog
WHERE Id="POPULAR_PRODUCT"
```

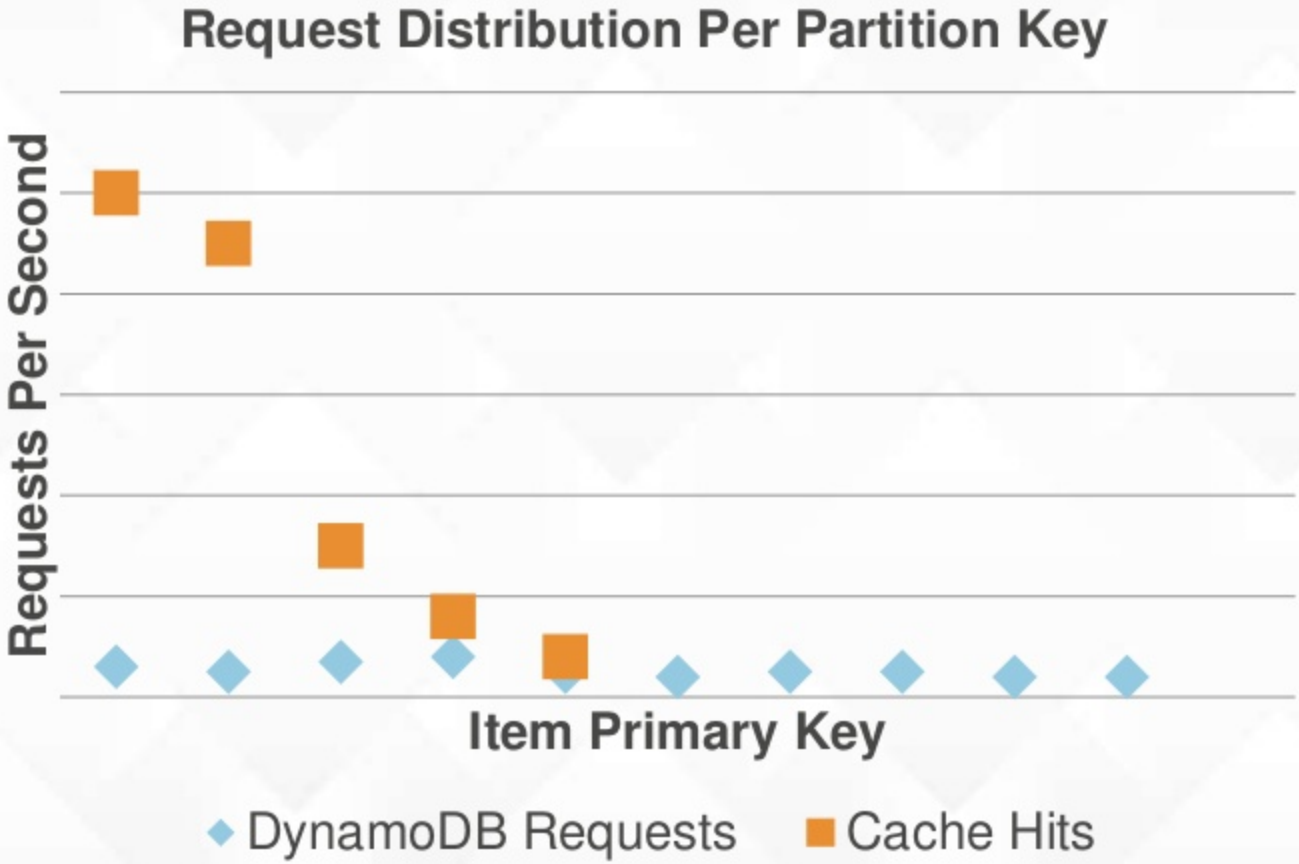



Request Distribution Per Partition Key
Requests Per Second
Item Primary Key
DynamoDB Requests

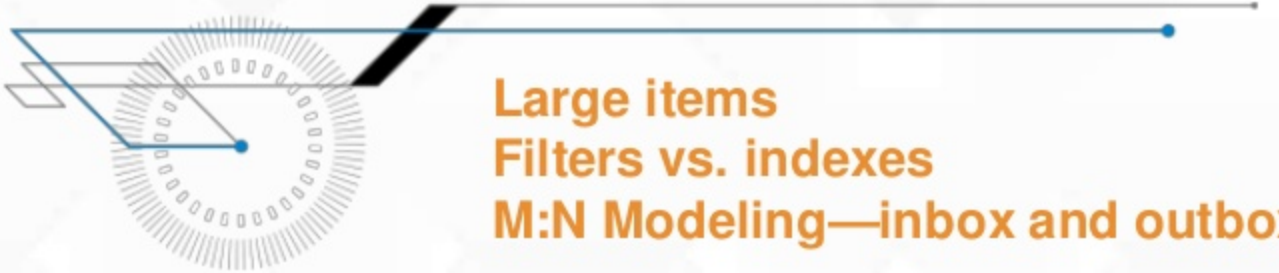
Cache popular items

```
SELECT Id, Description, ...
FROM ProductCatalog
WHERE Id="POPULAR_PRODUCT"
```




Request Distribution Per Partition Key
Requests Per Second
Item Primary Key
DynamoDB Requests
Cache Hits

Messaging App



Large items
Filters vs. indexes
M:N Modeling—inbox and outbox

Messages App

Inbox

```
SELECT *
FROM Messages
WHERE Recipient='David'
LIMIT 50
ORDER BY Date DESC
```

Messages Table

Outbox

```
SELECT *
FROM Messages
WHERE Sender ='David'
LIMIT 50
ORDER BY Date DESC
```

Large and small attributes mixed

Inbox

```
SELECT *
FROM Messages
WHERE Recipient='David'
LIMIT 50
ORDER BY Date DESC
```

Partition key

Sort key

Messages Table

Recipient	Date	Sender	Message
David	2014-10-02	Bob	…
… 48 more messages for David …			
David	2014-10-03	Alice	…
Alice	2014-09-28	Bob	…
Alice	2014-10-01	Carol	…

(Many more messages)

50 items × 256 KB each

Large message bodies
Attachments

Computing inbox query cost

$$50_{items} \times 256_{KB} \times \frac{1_{RCU}}{4_{KB}} \times \frac{1_{read}}{2_{e.c.reads}} = \mathbf{1600_{RCU}}$$

Items evaluated by query

Average item size

Conversion ratio

Eventually consistent reads

Separate the bulk data

Uniformly distributes large item reads

(50 sequential items at 128 bytes)

1. Query Inbox-GSI: 1 RCU
2. BatchGetItem Messages: 1600 RCU

(50 separate items at 256 KB)

Inbox-GSI

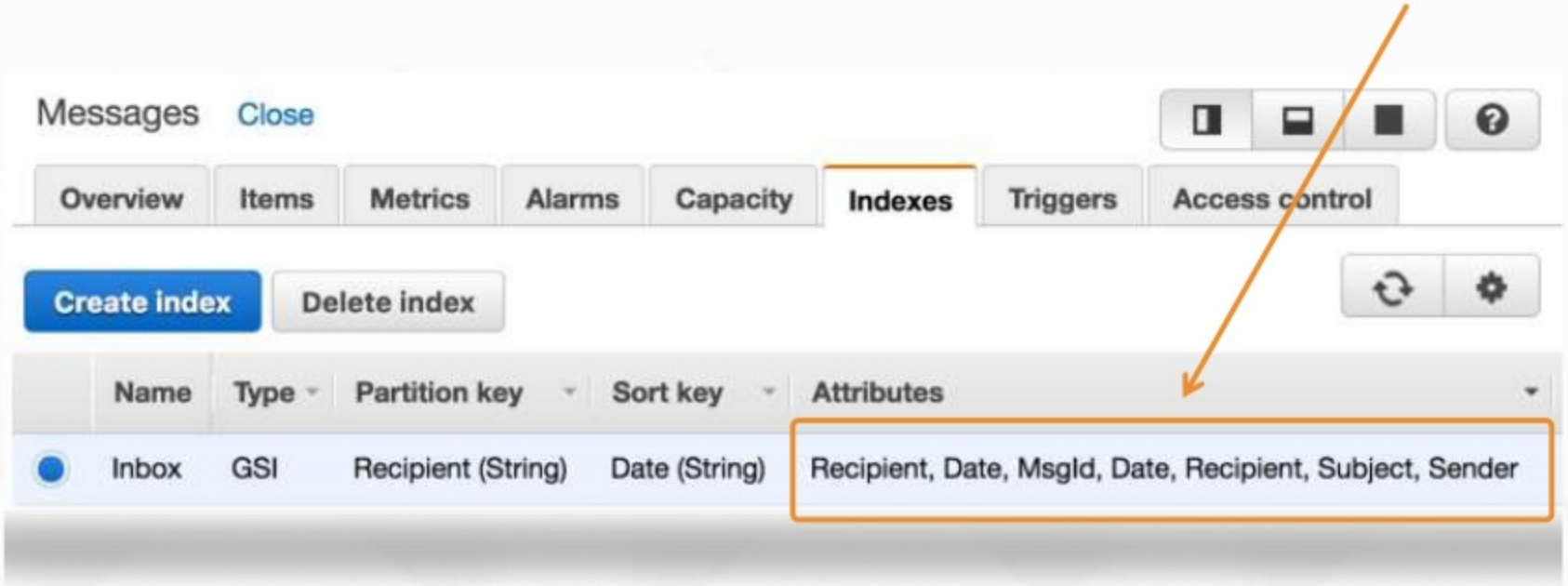
Recipient	Date	Sender	Subject	MsgId
David	2014-10-02	Bob	Hi!…	afed
David	2014-10-03	Alice	RE: The…	3kf8
Alice	2014-09-28	Bob	FW: Ok…	9d2b
Alice	2014-10-01	Carol	Hi!...	ct7r

Messages Table

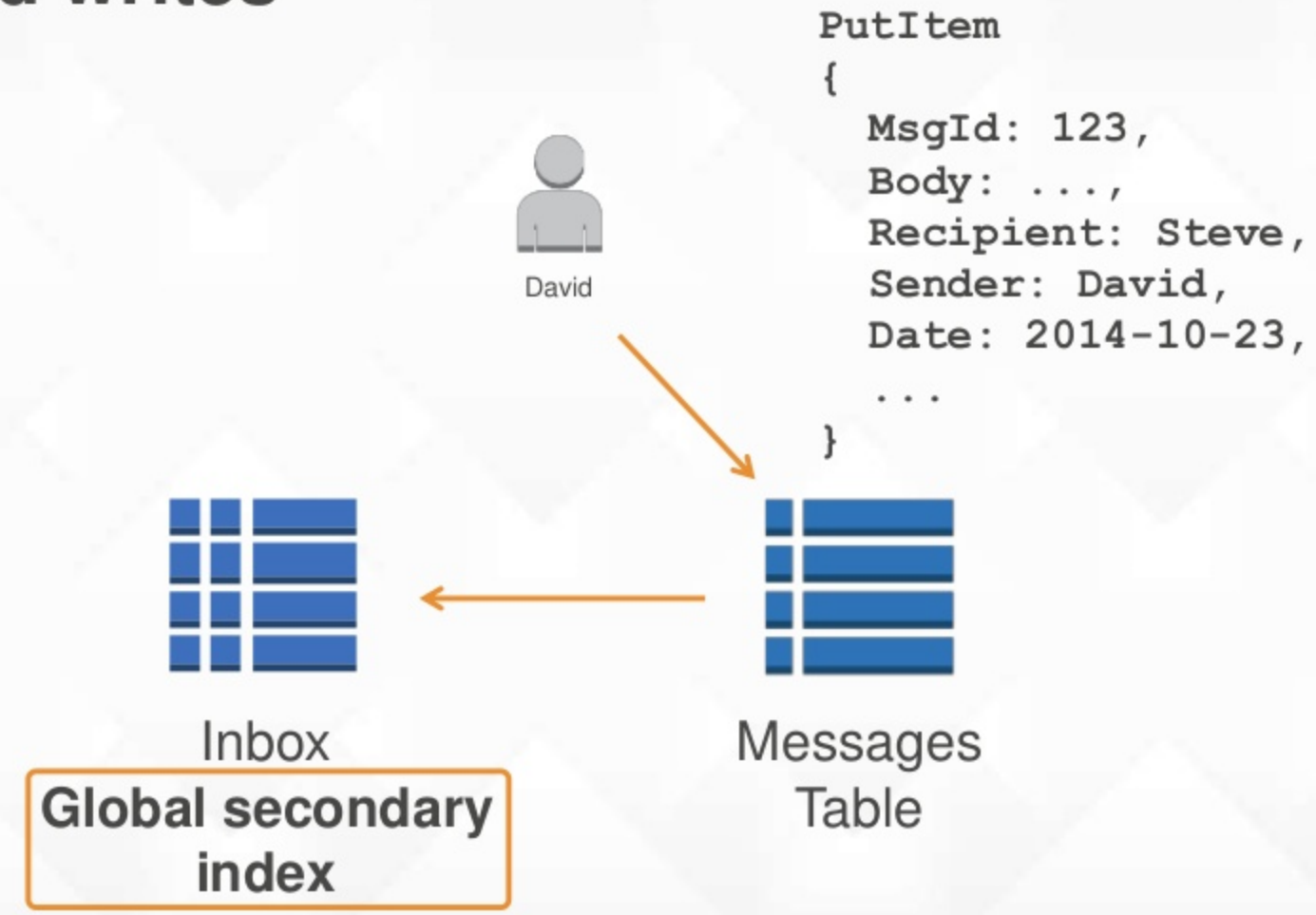
MsgId	Body
9d2b	…
3kf8	…
ct7r	…
afed	…

Inbox GSI

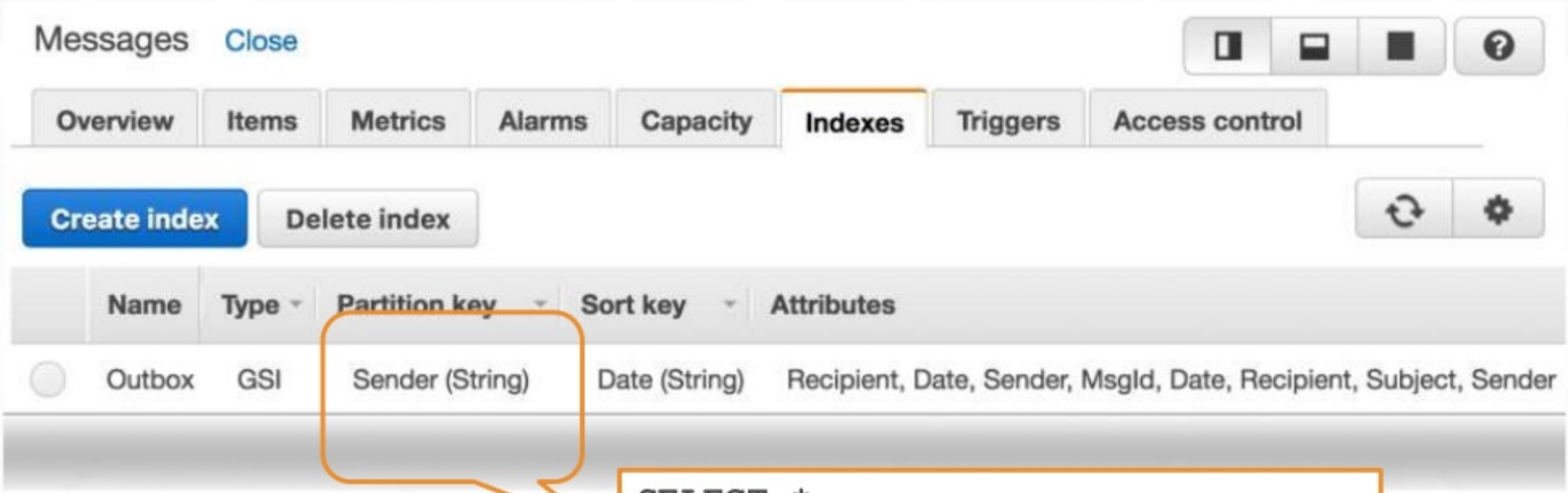
Define which attributes to copy into the index



Simplified writes

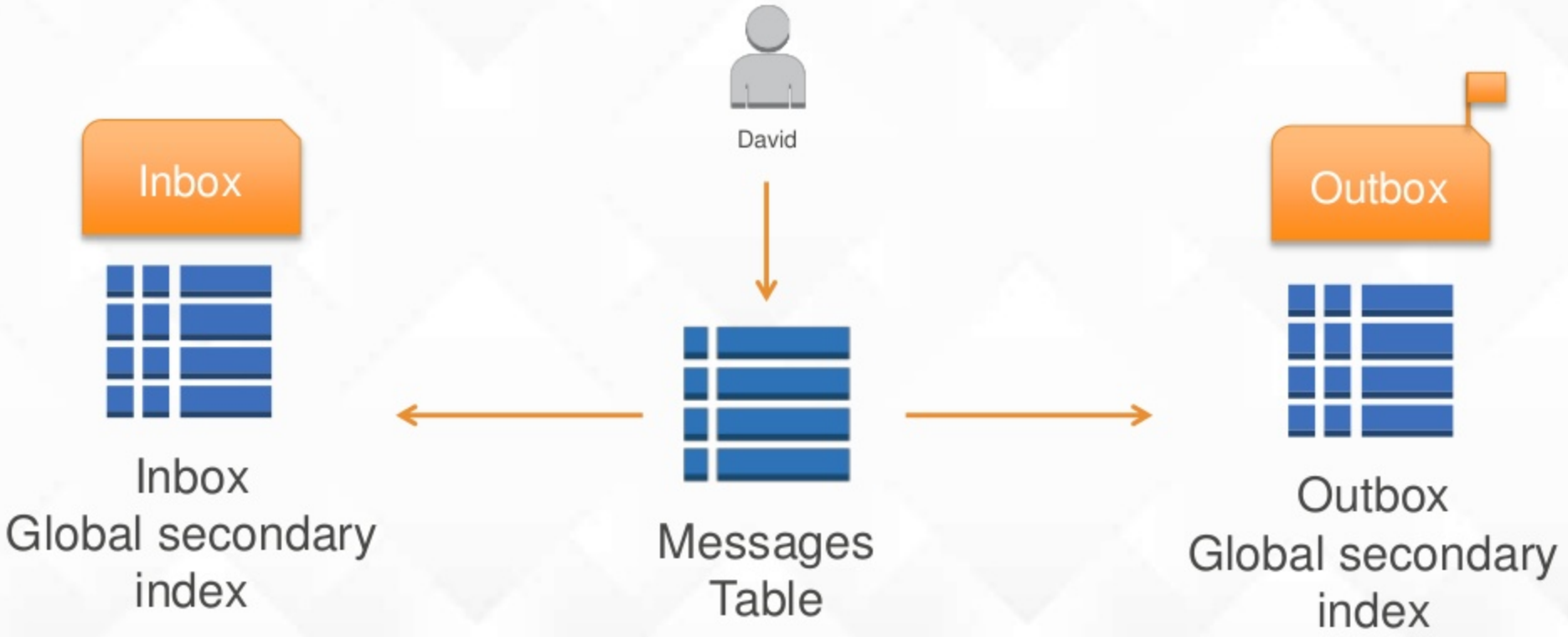


Outbox GSI



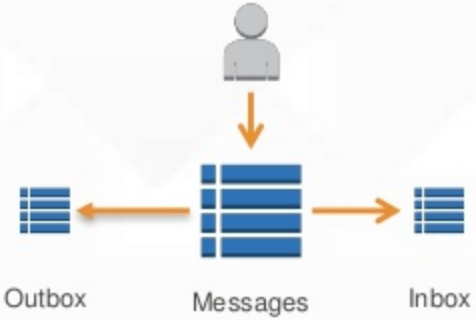

```
SELECT *
FROM Messages
WHERE Sender ='David'
LIMIT 50
ORDER BY Date DESC
```

Messaging app



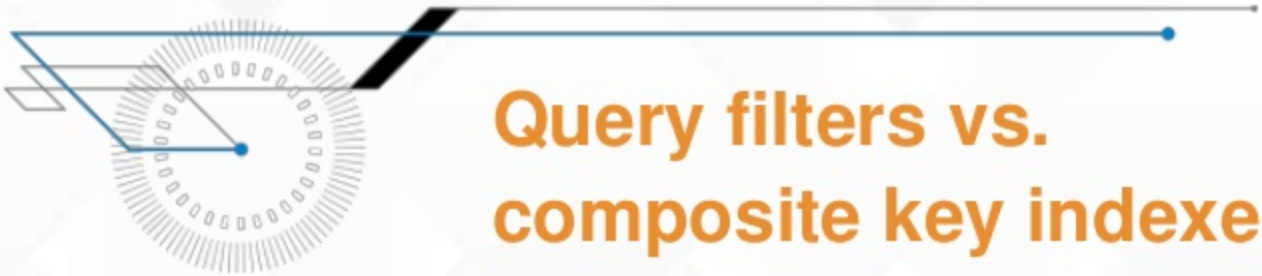
Distribute large items

- Reduce one-to-many item sizes
- Configure secondary index projections
- Use GSIs to model M:N relationship between sender and recipient



Important when: Querying many large items at once

Multiplayer Online Gaming

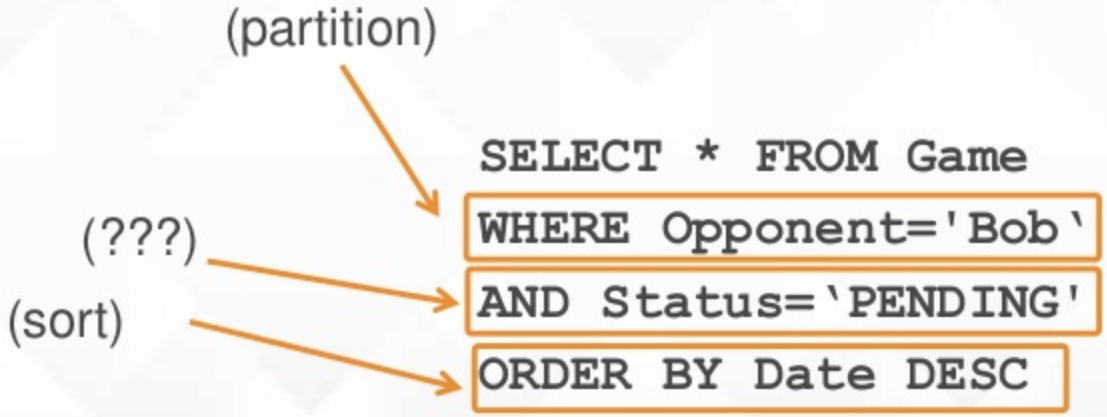


Query filters vs. composite key indexes

Multiplayer online game data

Query for incoming game requests

- DynamoDB indexes provide partition and sort key
- What about queries for two equalities and a sort?



Approach 1: Query filter

Partition key

Sort key

Secondary Index

Opponent	Date	GameId	Status	Host
Alice	2014-10-02	d9bl3	DONE	David
Carol	2014-10-08	o2pnb	IN_PROGRESS	Bob
Bob	2014-09-30	72f49	PENDING	Alice
Bob	2014-10-03	b932s	PENDING	Carol
Bob	2014-10-03	ef9ca	IN_PROGRESS	David

Approach 1: Query filter

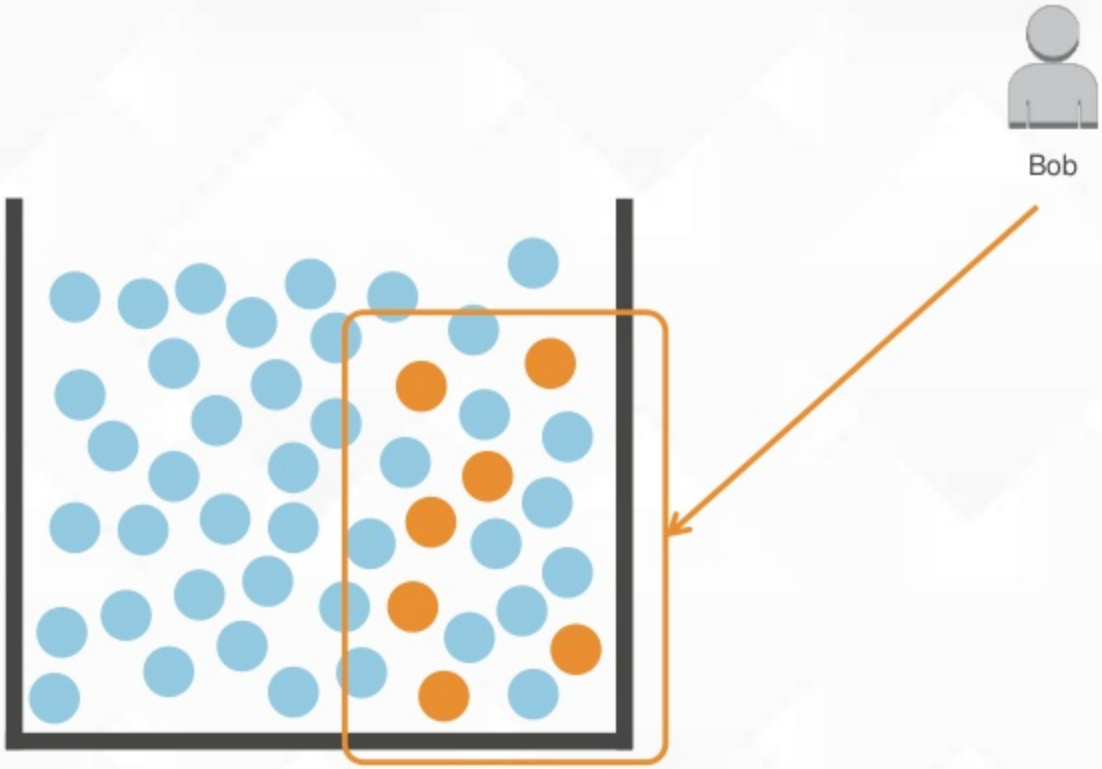
```
SELECT * FROM Game
WHERE Opponent='Bob'
ORDER BY Date DESC
FILTER ON Status='PENDING'
```

Secondary Index

Opponent	Date	GameId	Status	Host
Alice	2014-10-02	d9bl3	DONE	David
Carol	2014-10-08	o2pnb	IN_PROGRESS	Bob
Bob	2014-09-30	72f49	PENDING	Alice
Bob	2014-10-03	b932s	PENDING	Carol
Bob	2014-10-03	ef9ca	IN_PROGRESS	David

(filtered out)

Needle in a haystack



Use query filter

- Send back less data “on the wire”
- Simplify application code
- Simple SQL-like expressions
  - AND, OR, NOT, ()

Important when: Your index isn’t entirely selective

Approach 2: Composite key

Status
DONE
IN_PROGRESS
IN_PROGRESS
PENDING
PENDING

+

Date
2014-10-02
2014-10-08
2014-10-03
2014-10-03
2014-09-30

=

StatusDate
DONE_2014-10-02
IN_PROGRESS_2014-10-08
IN_PROGRESS_2014-10-03
PENDING_2014-09-30
PENDING_2014-10-03

Approach 2: Composite key

Partition key

Sort key

Secondary Index

Opponent	StatusDate	GameId	Host
Alice	DONE_2014-10-02	d9bl3	David
Carol	IN_PROGRESS_2014-10-08	o2pnb	Bob
Bob	IN_PROGRESS_2014-10-03	ef9ca	David
Bob	PENDING_2014-09-30	72f49	Alice
Bob	PENDING_2014-10-03	b932s	Carol

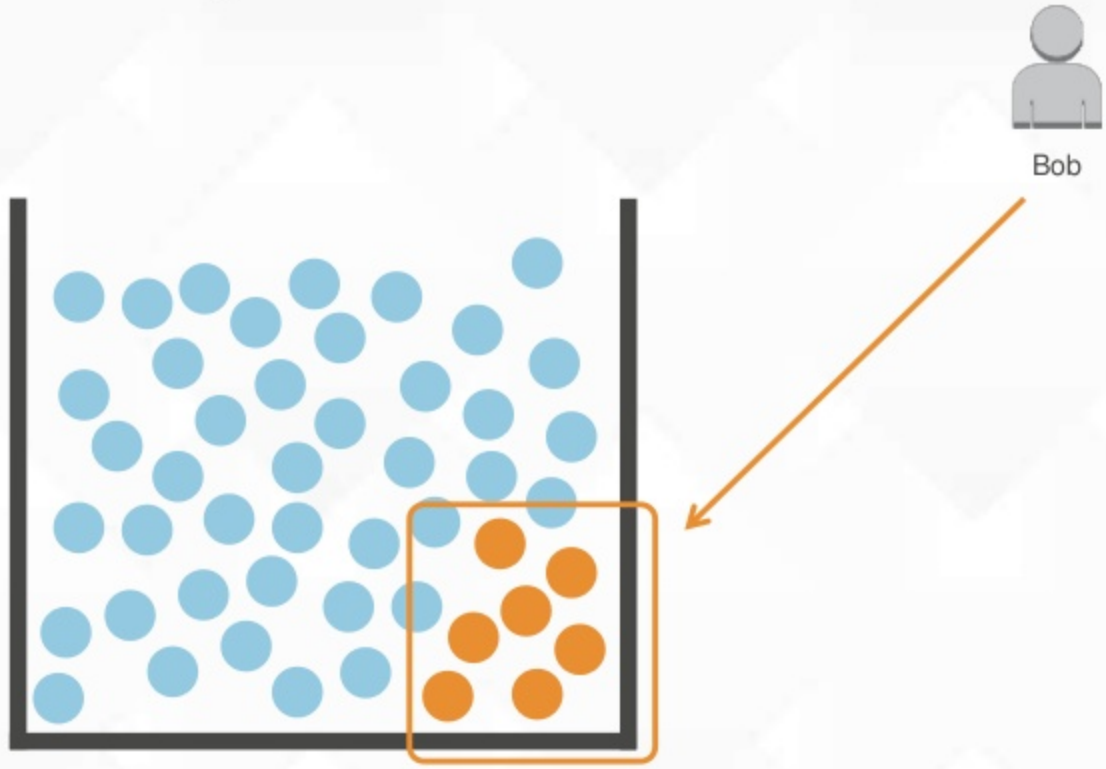
Approach 2: Composite key

```
SELECT * FROM Game
WHERE Opponent='Bob'
    AND StatusDate BEGINS_WITH 'PENDING'
```

Secondary Index

Opponent	StatusDate	GameId	Host
Alice	DONE_2014-10-02	d9bl3	David
Carol	IN_PROGRESS_2014-10-08	o2pnb	Bob
Bob	IN_PROGRESS_2014-10-03	ef9ca	David
Bob	PENDING_2014-09-30	72f49	Alice
Bob	PENDING_2014-10-03	b932s	Carol

Needle in a *sorted* haystack



Sparse indexes

Scan sparse partition GSIs

Game-scores-table

Id (Part.)	User	Game	Score	Date	Award
1	Bob	G1	1300	2012-12-23	
2	Bob	G1	1450	2012-12-23	
3	Jay	G1	1600	2012-12-24	
4	Mary	G1	2000	2012-10-24	Champ
5	Ryan	G2	123	2012-03-10	
6	Jones	G2	345	2012-03-20	

Award-GSI

Award (Part.)	Id	User	Score
Champ	4	Mary	2000

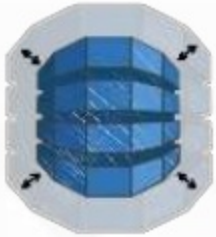
Replace filter with indexes

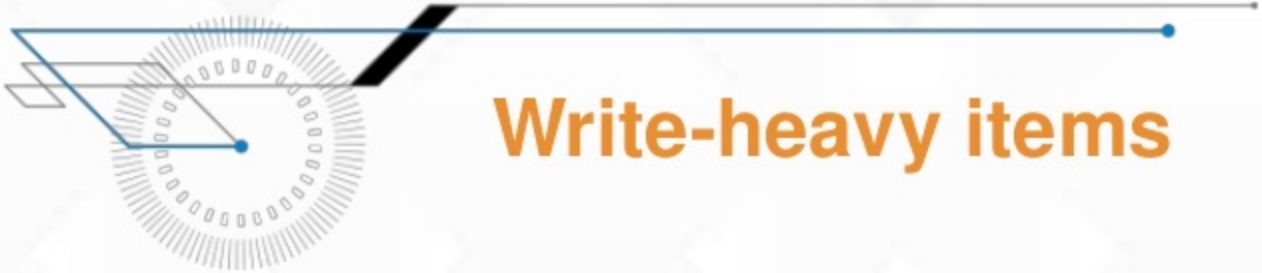
- Concatenate attributes to form useful secondary index keys

- Take advantage of sparse indexes

Important when: You want to optimize a query as much as possible

Real-Time Voting



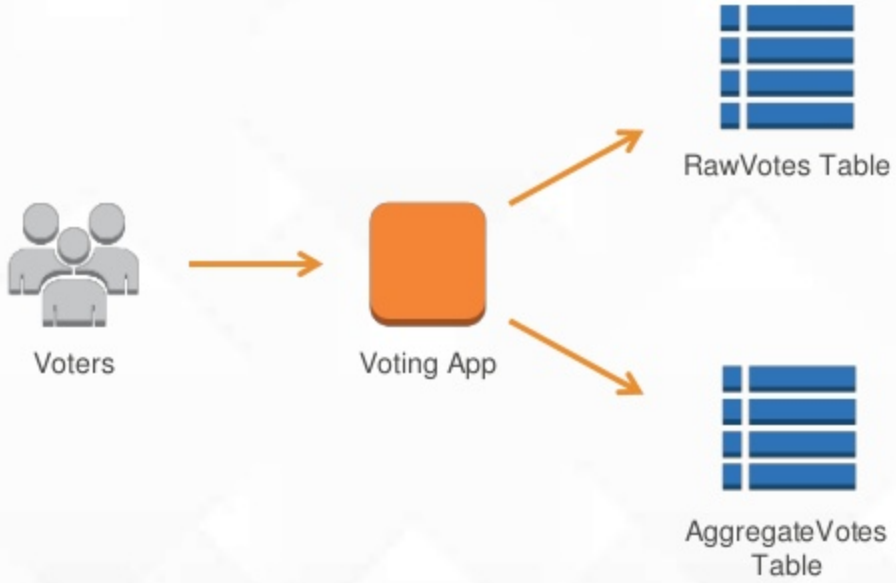


Write-heavy items

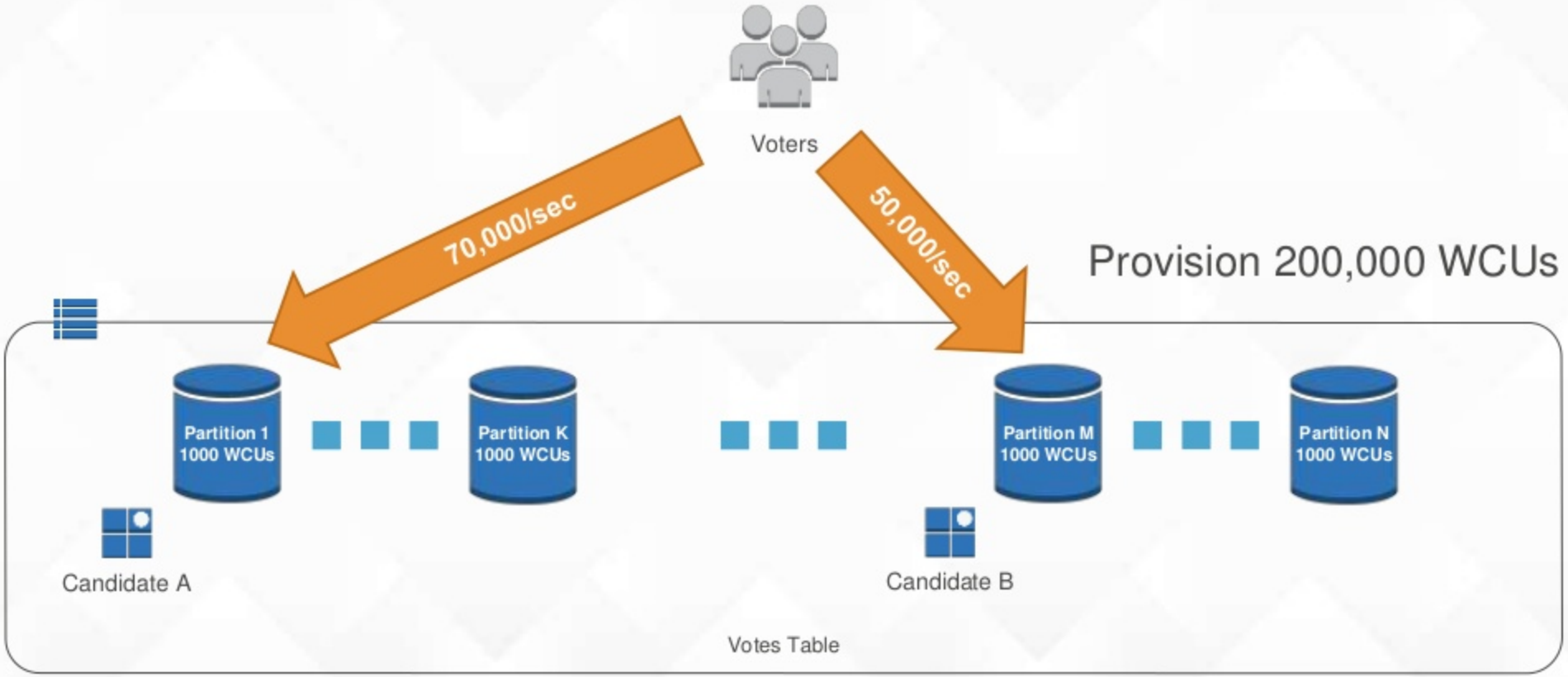
Requirements for voting

- Allow each person to vote only once
- No changing votes
- Real-time aggregation
- Voter analytics, demographics

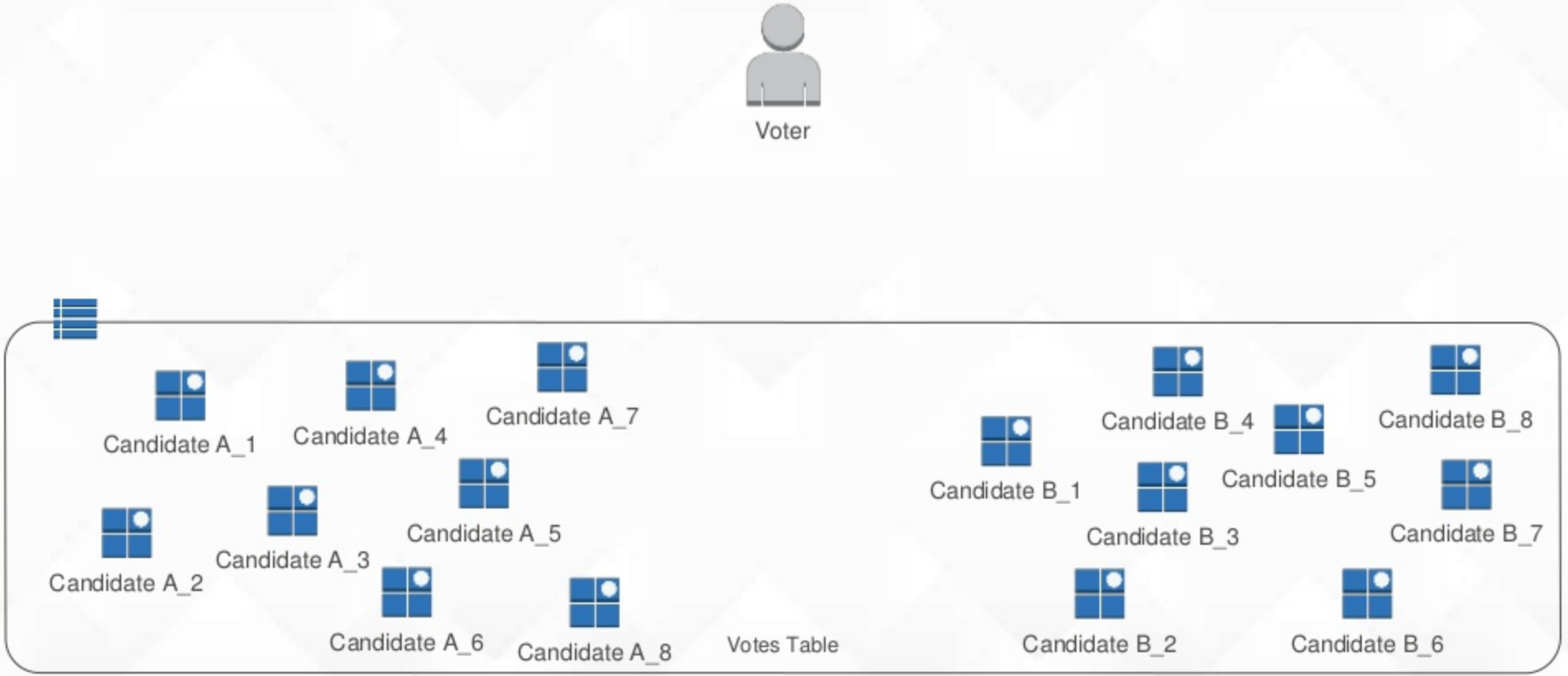
Real-time voting architecture



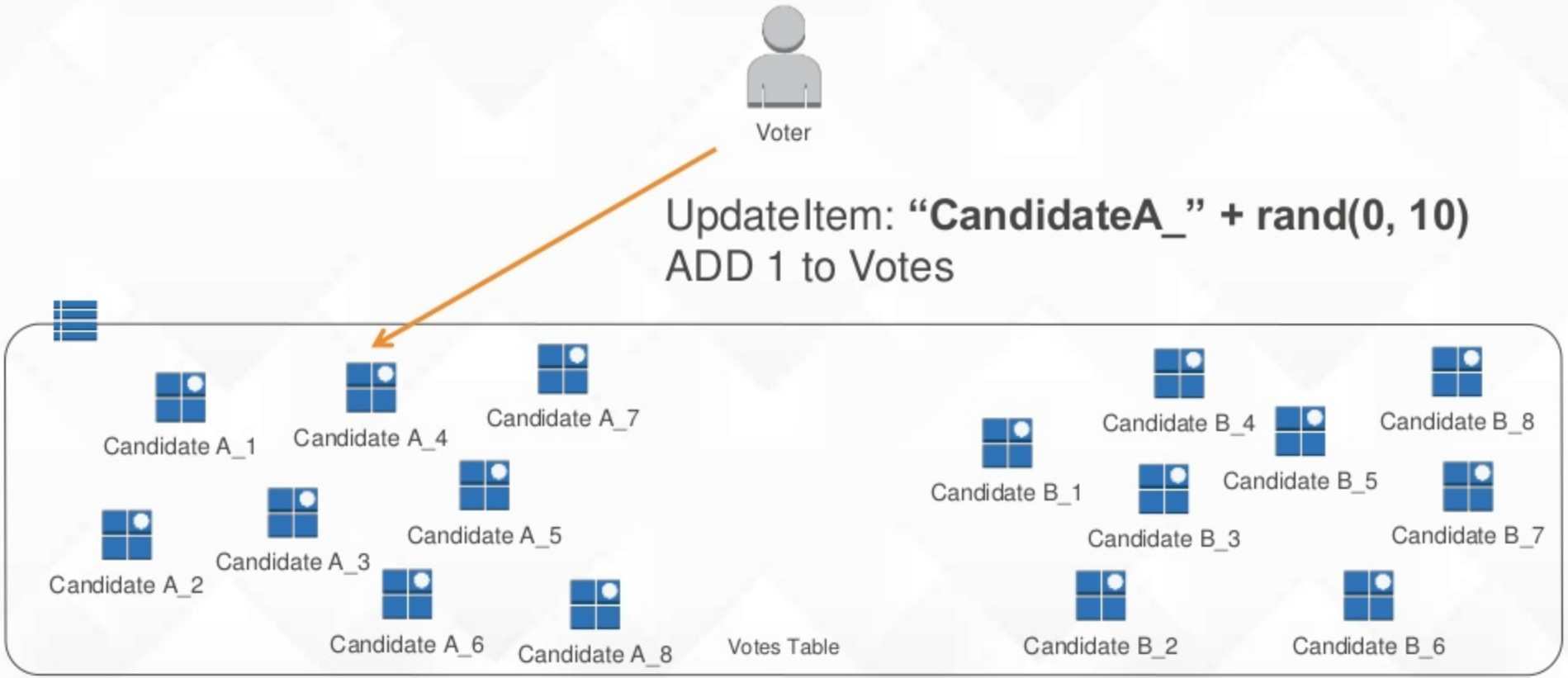
Scaling bottlenecks



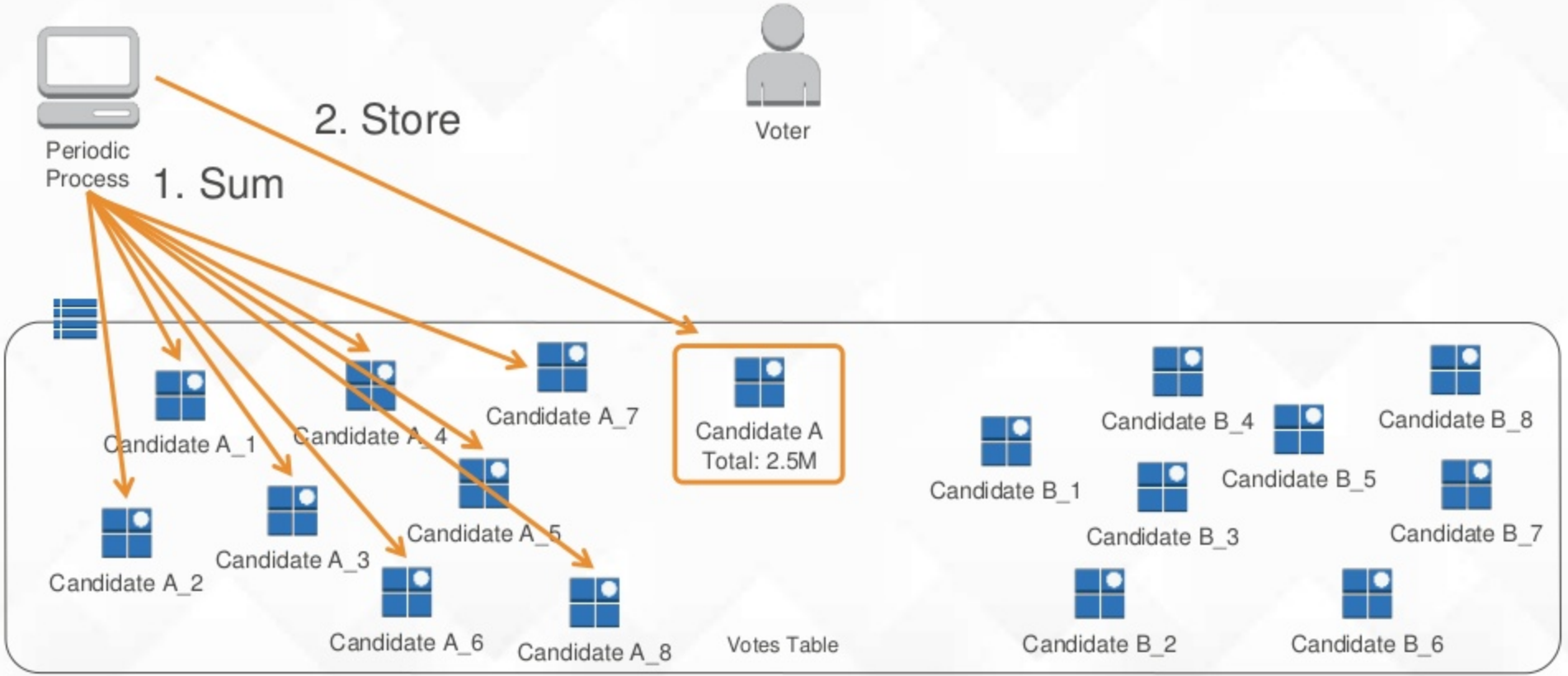
Write sharding



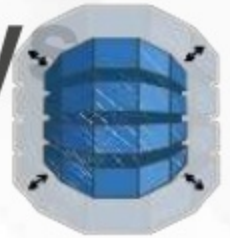
Write sharding



Shard aggregation



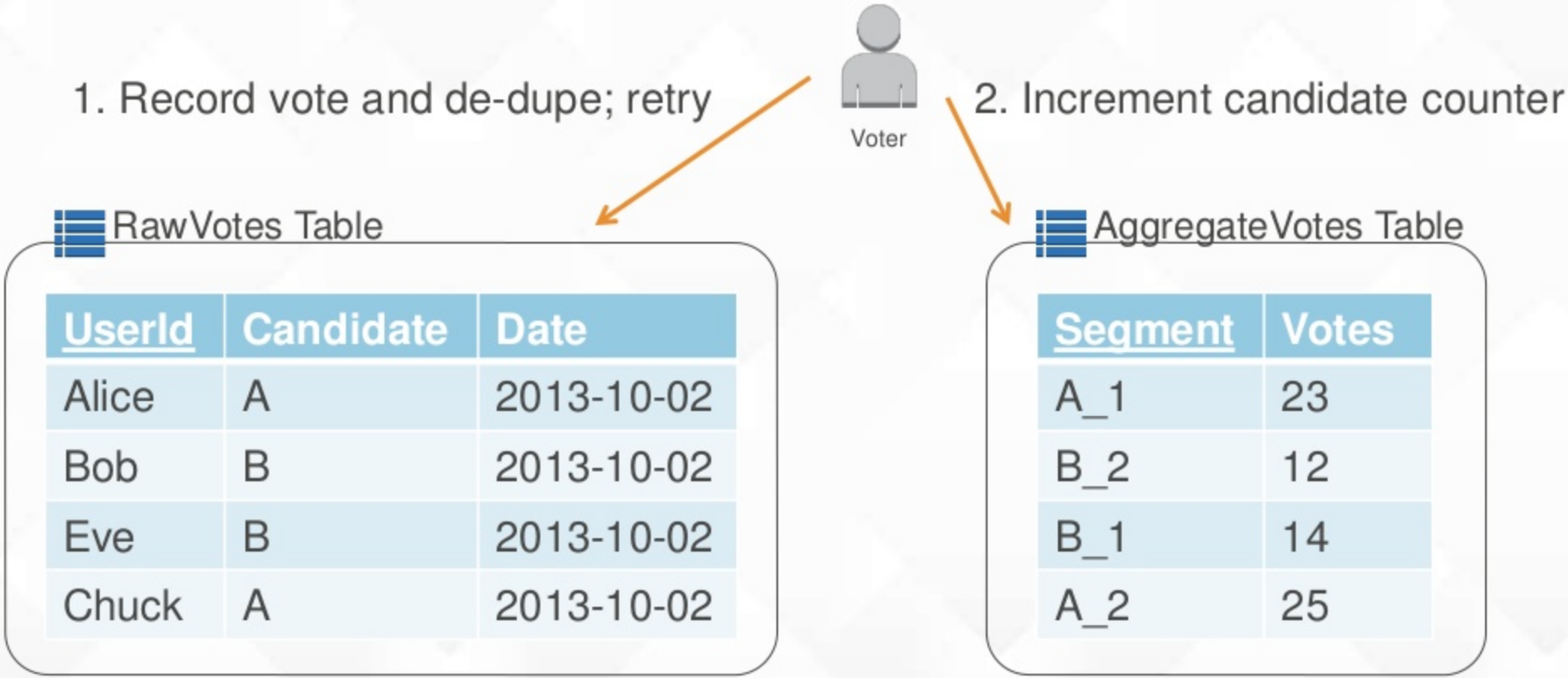
Shard write-heavy partition key



- Trade off read cost for write scalability
- Consider throughput per partition key and per partition

Important when: Your write workload is not horizontally scalable

Correctness in voting



Correctness in aggregation?

Voter

RawVotes Table

UserId	Candidate	Date
Alice	A	2013-10-02
Bob	B	2013-10-02
Eve	B	2013-10-02
Chuck	A	2013-10-02

AggregateVotes Table

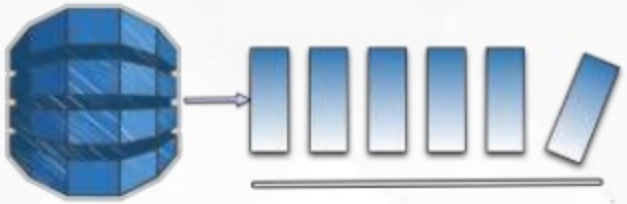
Segment	Votes
A_1	23
B_2	12
B_1	14
A_2	25

DynamoDB Streams

DynamoDB Streams

- Stream of updates to a table
- Asynchronous
- Exactly once
- Strictly ordered
  - Per item
- Highly durable
- Scale with table
- 24-hour lifetime
- Sub-second latency

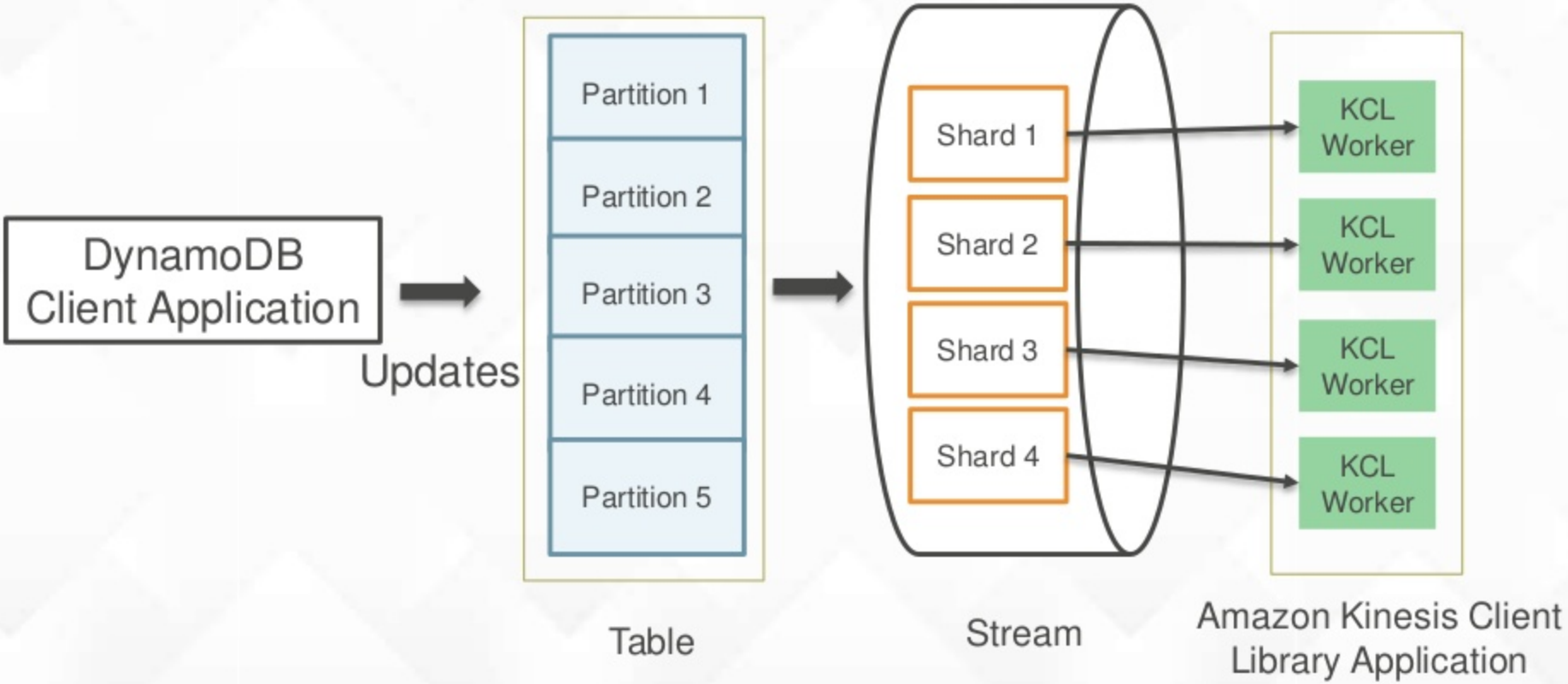
View types



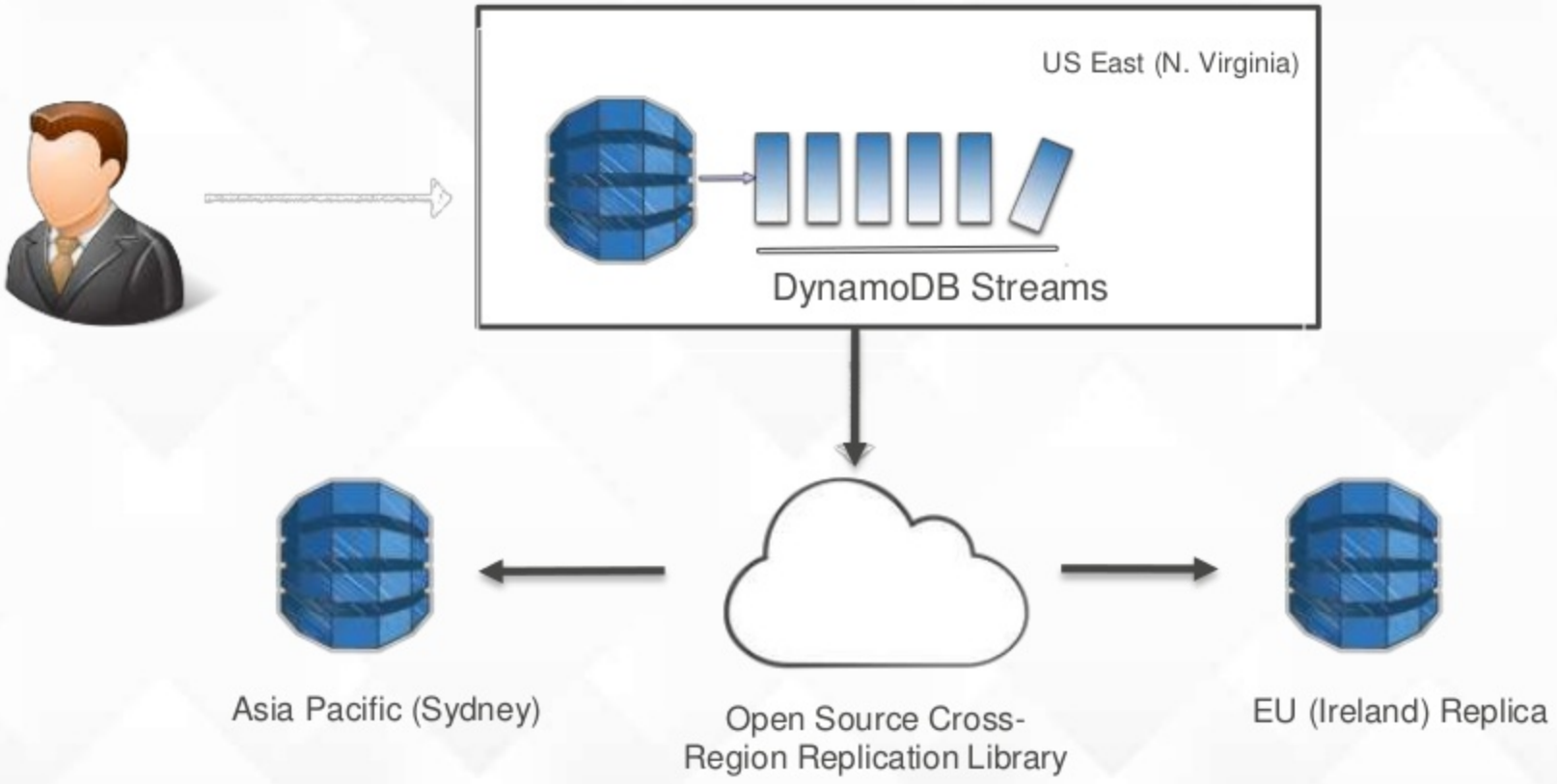
UpdateItem (Name = John, Destination = Pluto)

View Type	Destination
Old image—before update	Name = John, Destination = Mars
New image—after update	Name = John, Destination = Pluto
Old and new images	Name = John, Destination = Mars Name = John, Destination = Pluto
Keys only	Name = John

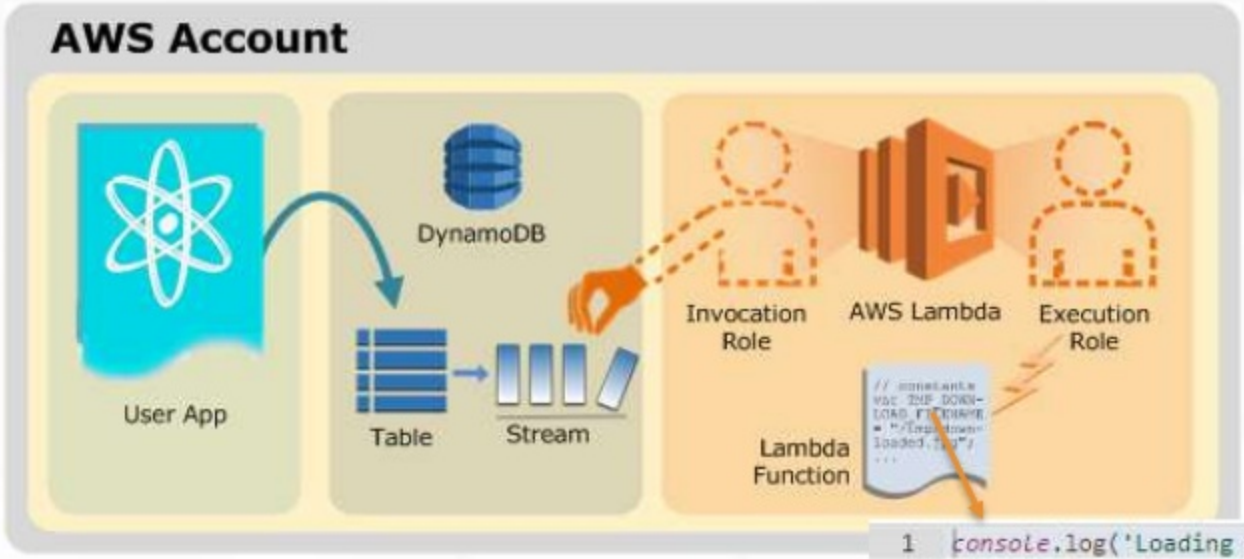
DynamoDB Streams and Amazon Kinesis Client Library



Cross-region replication



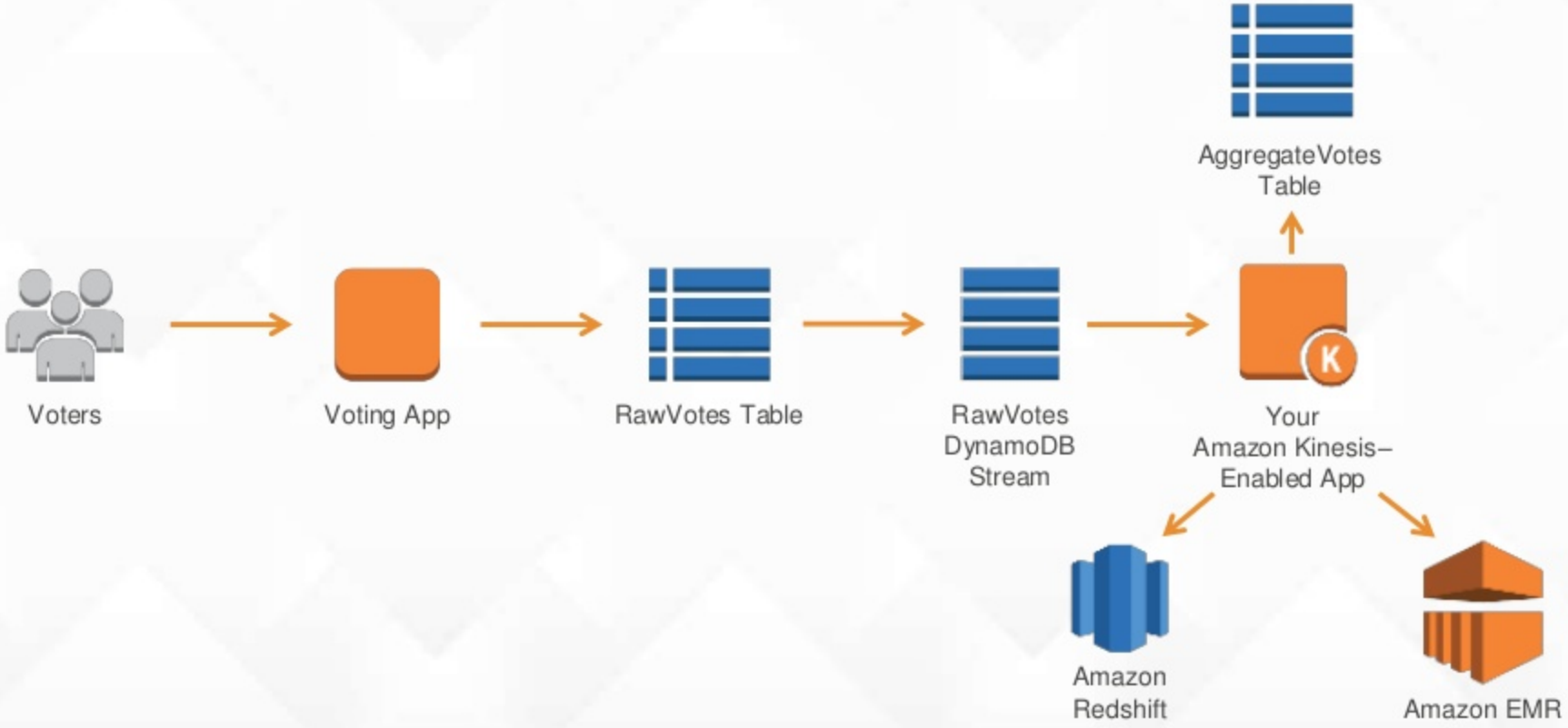
DynamoDB Streams and AWS Lambda




```
1  console.log('Loading event');
2  exports.handler = function(event, context) {
3    console.log("Event: %j", event);
4    for(i = 0; i < event.Records.length; ++i) {
5      record = event.Records[i];
6      console.log(record.EventID);
7      console.log(record.EventName);
8      console.log("DynamoDB Record: %j", record.Dynamodb);
9    }
10   context.done(null, "Hello World");  // SUCCESS with message
11 }
```

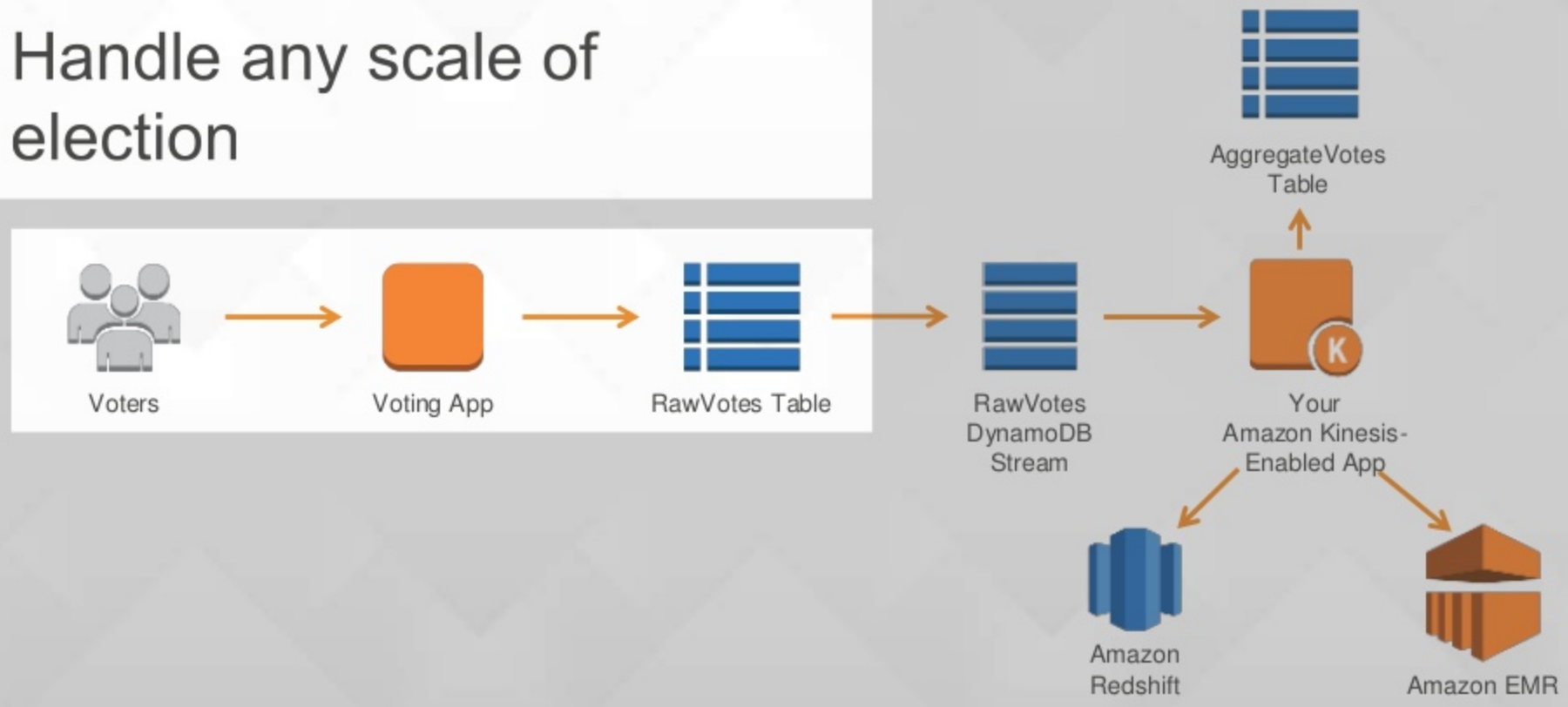
```
2015-03-21T07:44:58.883Z 2ca3769a-cf9e-11e4-b270-ad4d24b312ff INSERT
2015-03-21T07:44:58.883Z 2ca3769a-cf9e-11e4-b270-ad4d24b312ff DynamoDB Record:{ "NewImage": { "name": { "S": "sivar" }, "hk": { "S": "3" } }, "SizeBytes": 15, "StreamViewType": "NEW_AND_OLD_IMAG
2015-03-21T07:44:58.883Z 2ca3769a-cf9e-11e4-b270-ad4d24b312ff Message: "Hello World"
```

Real-time voting architecture (improved)



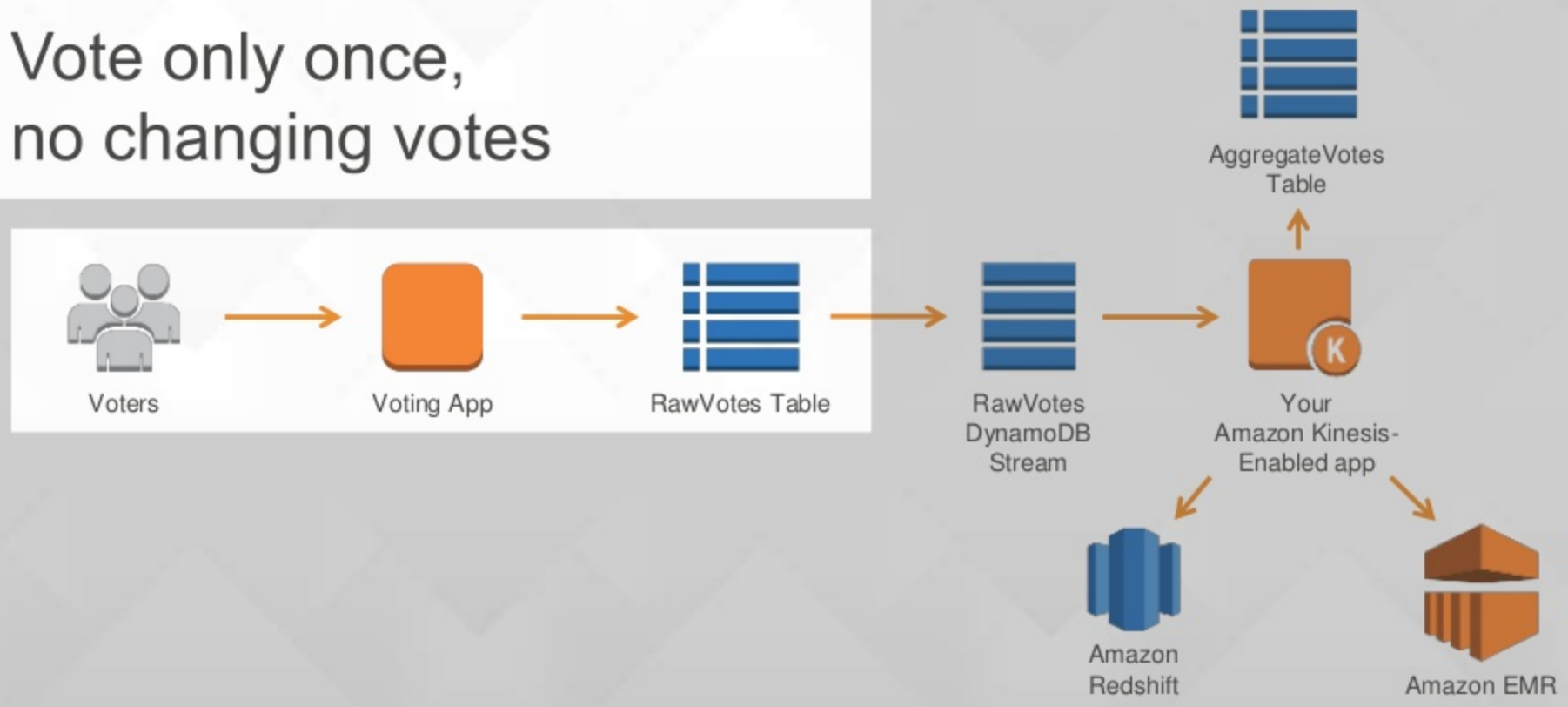
Real-time voting architecture

Handle any scale of election



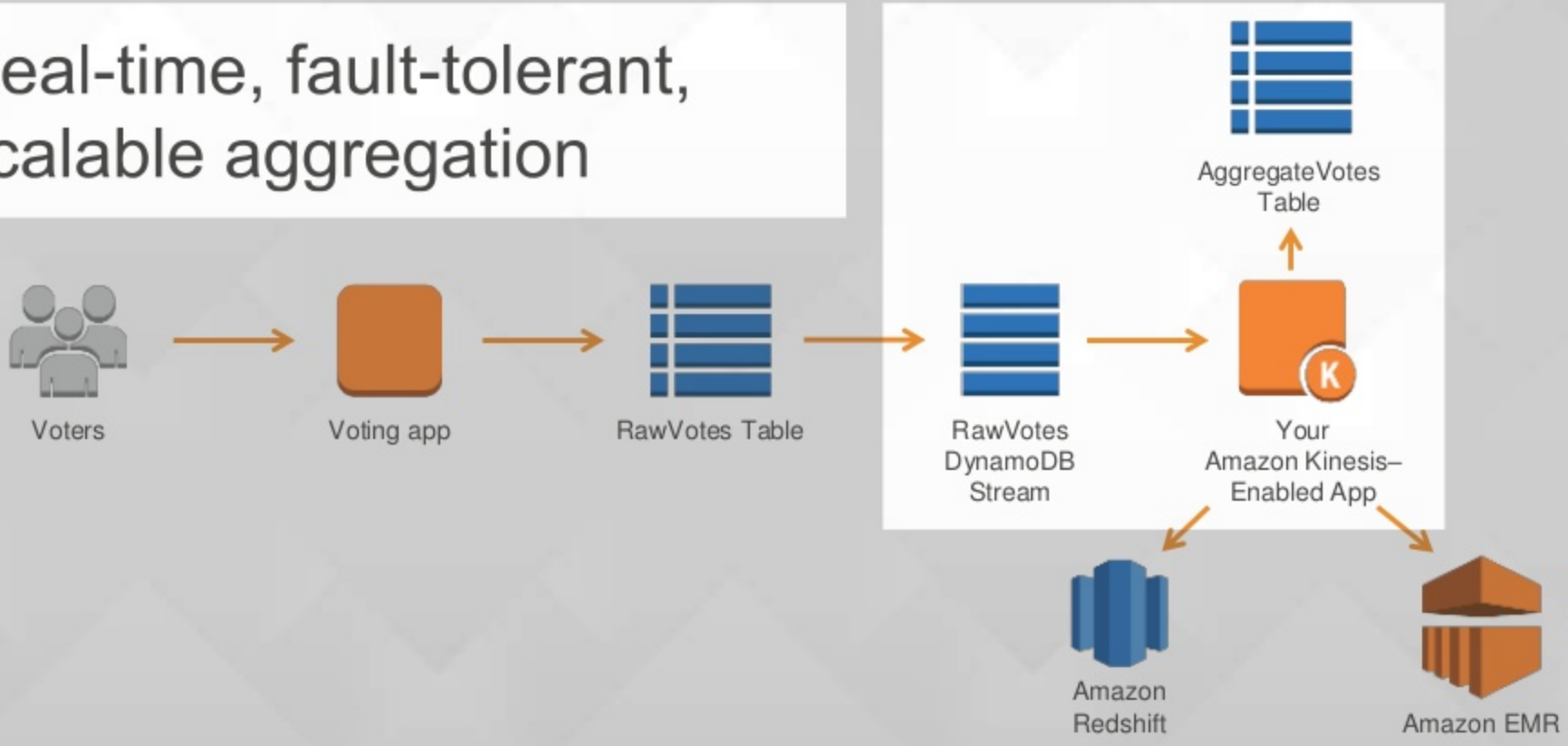
Real-time voting architecture

Vote only once,
no changing votes



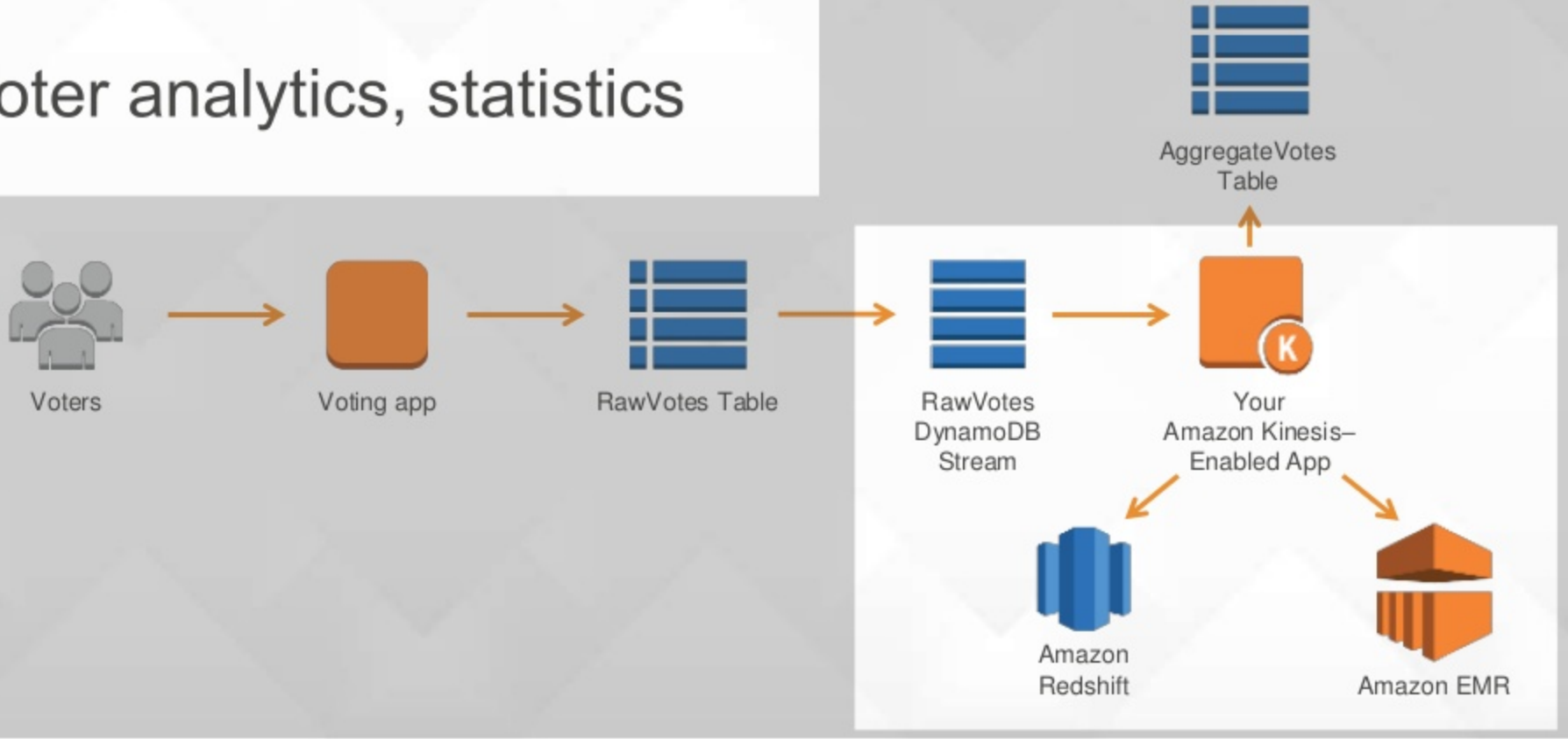
Real-time voting architecture

Real-time, fault-tolerant, scalable aggregation

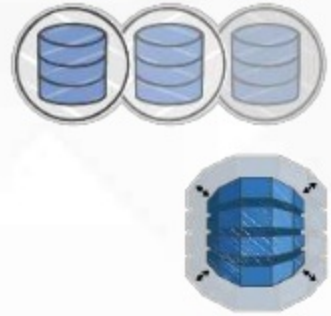


Real-time voting architecture

Voter analytics, statistics



Analytics with DynamoDB Streams



- Collect and de-dupe data in DynamoDB
- Aggregate data in-memory and flush periodically

Important when: Performing real-time aggregation and analytics

Architecture

Reference Architecture