DynamoDB

Design Patterns and Best Practices

Rick Houlihan, Principal Solutions Architect

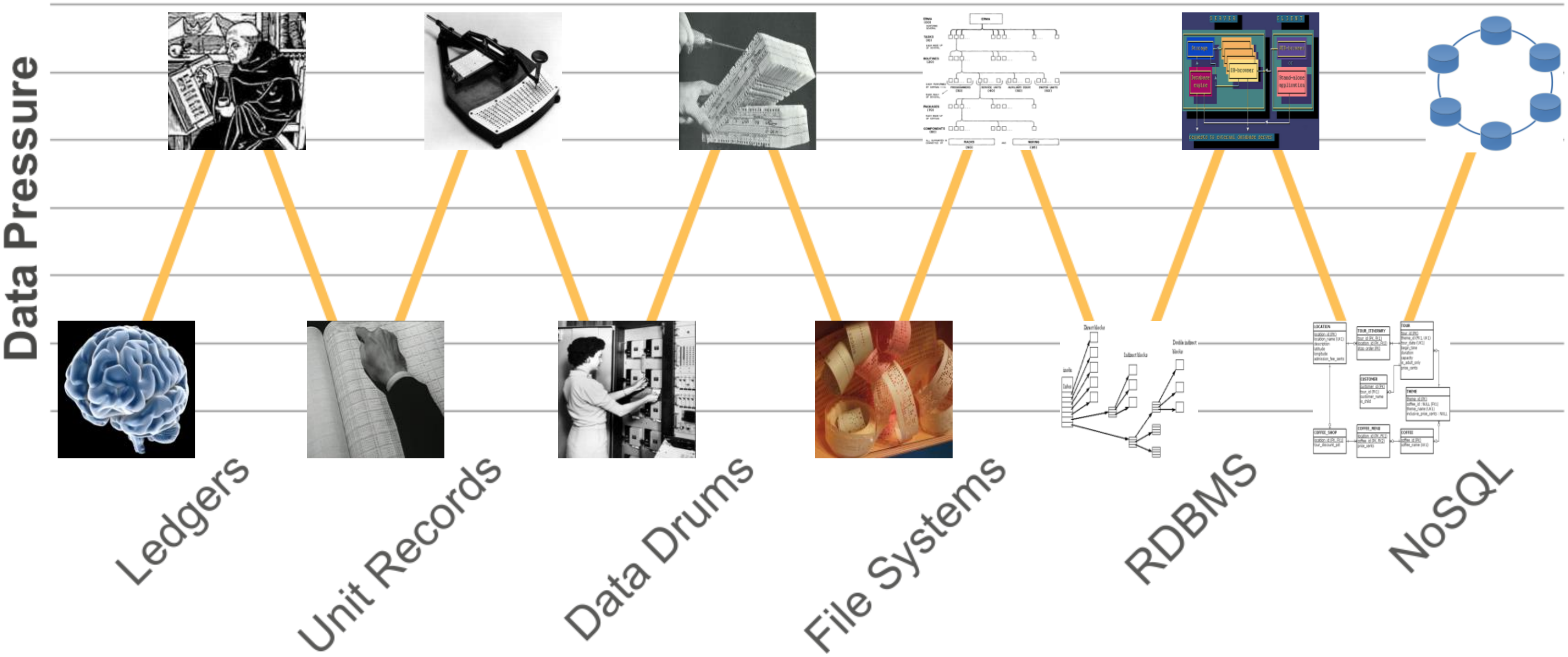
1/20/2016

amazon web services | Webinars

What to expect from the session

- Brief history of data processing
- DynamoDB Internals
    - Tables, API, data types, indexes
    - Scaling and data modeling
- Design patterns and best practices
- Event driven applications and DDB Streams

Timeline of Database Technology



Data Pressure

Ledgers

Unit Records

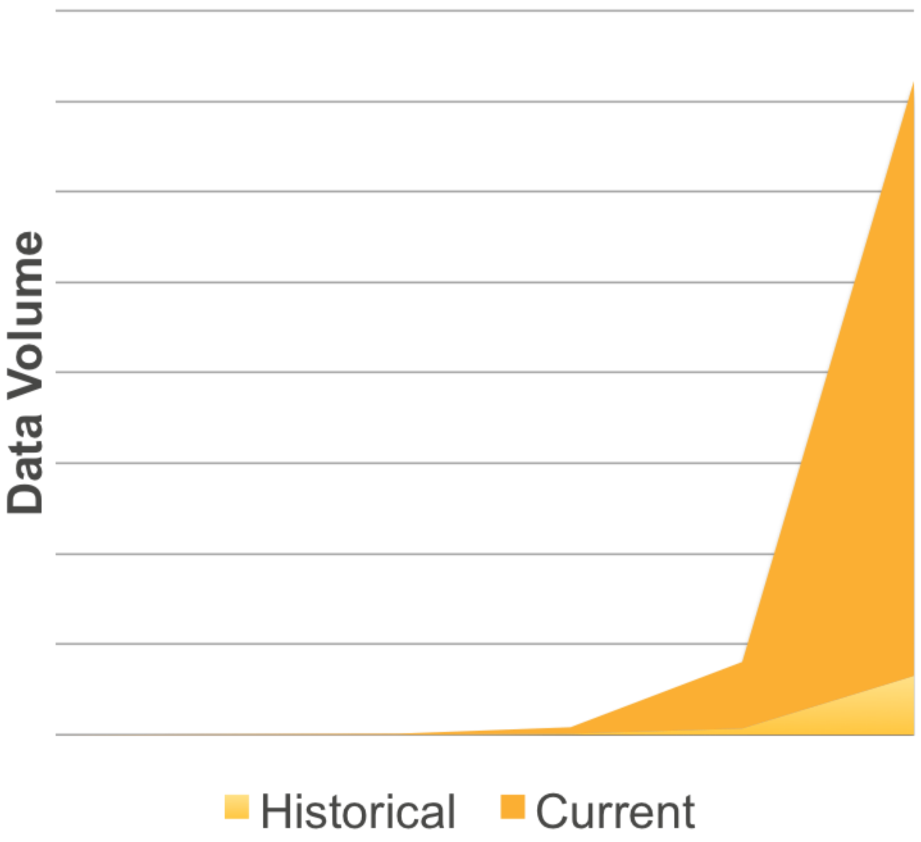
Data Drums

File Systems

RDBMS

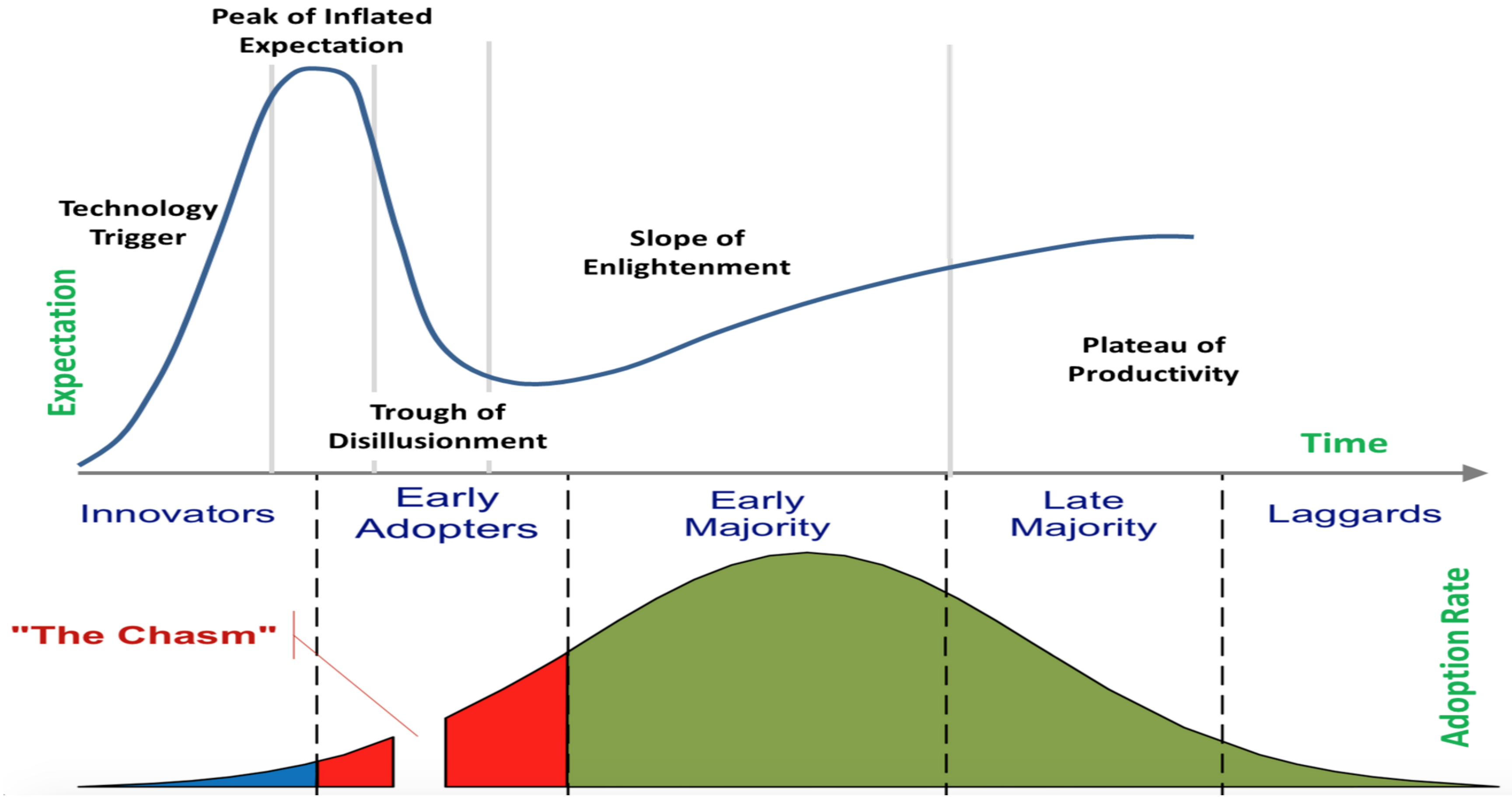
NoSQL

Data Volume Since 2010



- 90% of stored data generated in last 2 years
- 1 Terabyte of data in 2010 equals 6.5 Petabytes today
- Linear correlation between data pressure and technical innovation
- No reason these trends will not continue over time

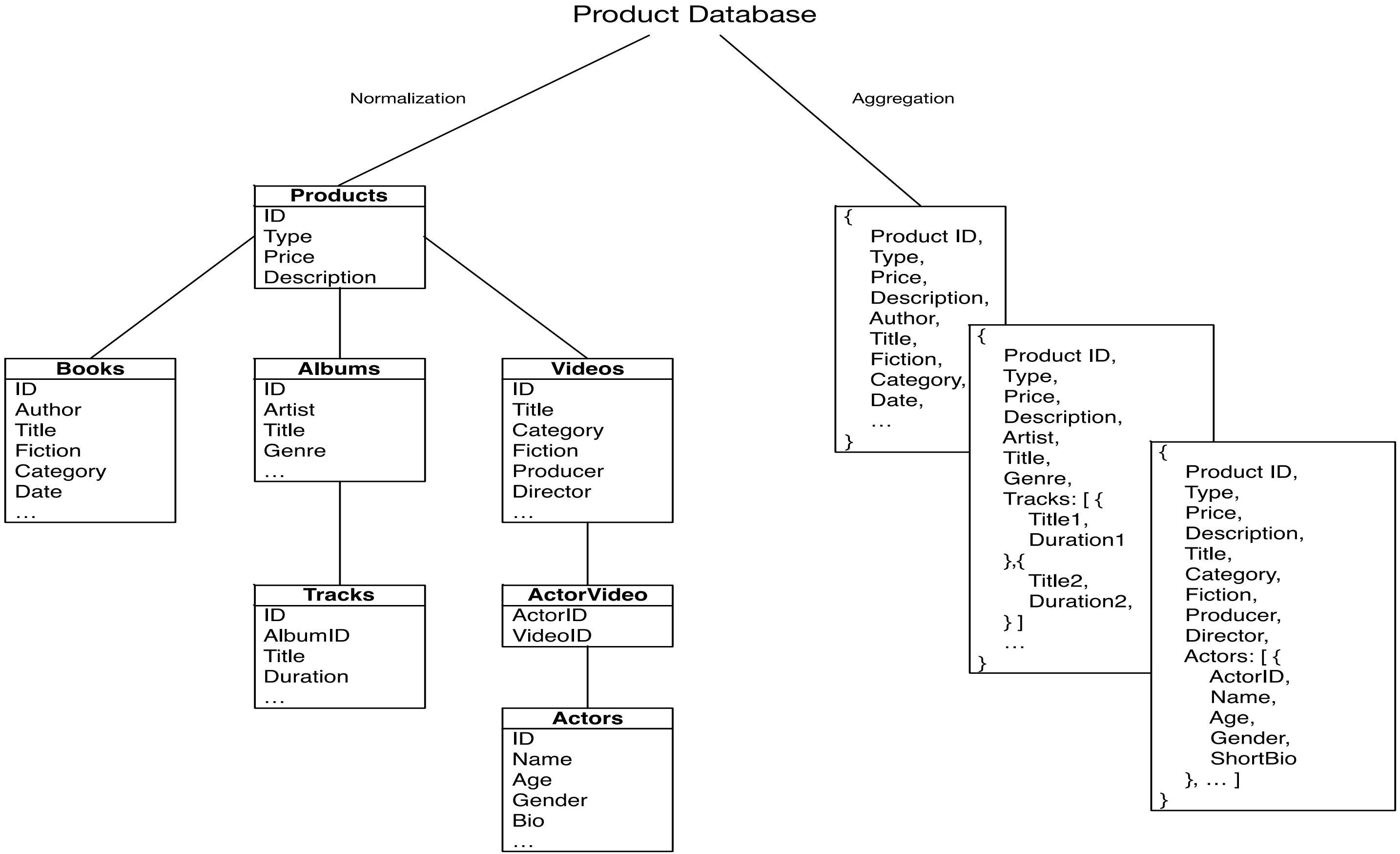
Technology Adoption and the Hype Curve



Why NoSQL?

SQL	NoSQL
Optimized for storage	**Optimized for compute**
Normalized/relational	Denormalized/hierarchical
Ad hoc queries	Instantiated views
Scale vertically	Scale horizontally
Good for OLAP	Built for OLTP at scale

SQL vs. NoSQL Access Pattern



Product Database

Normalization

Aggregation

Products
ID
Type
Price
Description

Books
ID
Author
Title
Fiction
Category
Date
…

Albums
ID
Artist
Title
Genre
…

Videos
ID
Title
Category
Fiction
Producer
Director
…

Tracks
ID
AlbumID
Title
Duration
…

ActorVideo
ActorID
VideoID

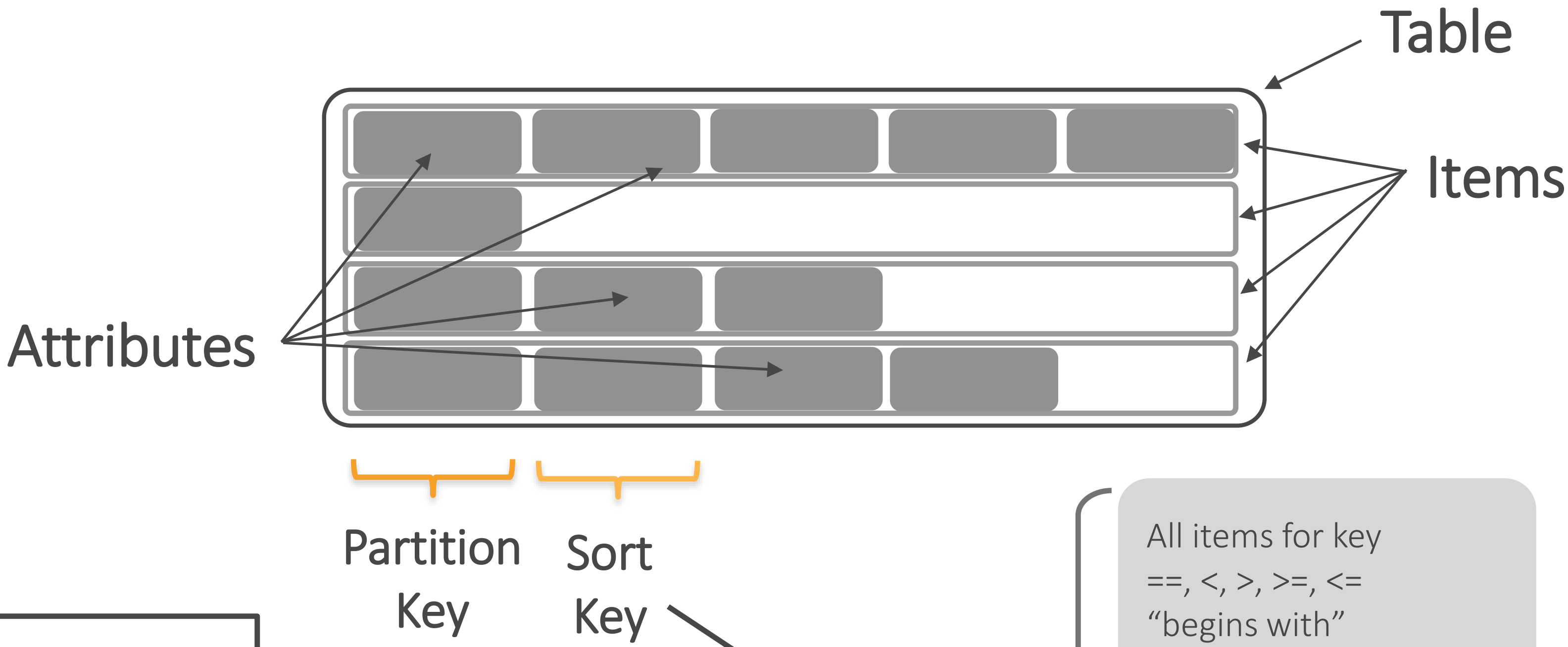
Actors
ID
Name
Age
Gender
Bio
…

```
{
   Product ID,
   Type,
   Price,
   Description,
   Author,
   Title,
   Fiction,
   Category,
   Date,
   …
}
```

```
{
   Product ID,
   Type,
   Price,
   Description,
   Artist,
   Title,
   Genre,
   Tracks: [ {
      Title1,
      Duration1
   },{
      Title2,
      Duration2,
   } ]
   …
}
```

```
{
   Product ID,
   Type,
   Price,
   Description,
   Title,
   Category,
   Fiction,
   Producer,
   Director,
   Actors: [ {
      ActorID,
      Name,
      Age,
      Gender,
      ShortBio
   }, … ]
}
```

Table



Table

Items

Attributes

Partition Key

Sort Key

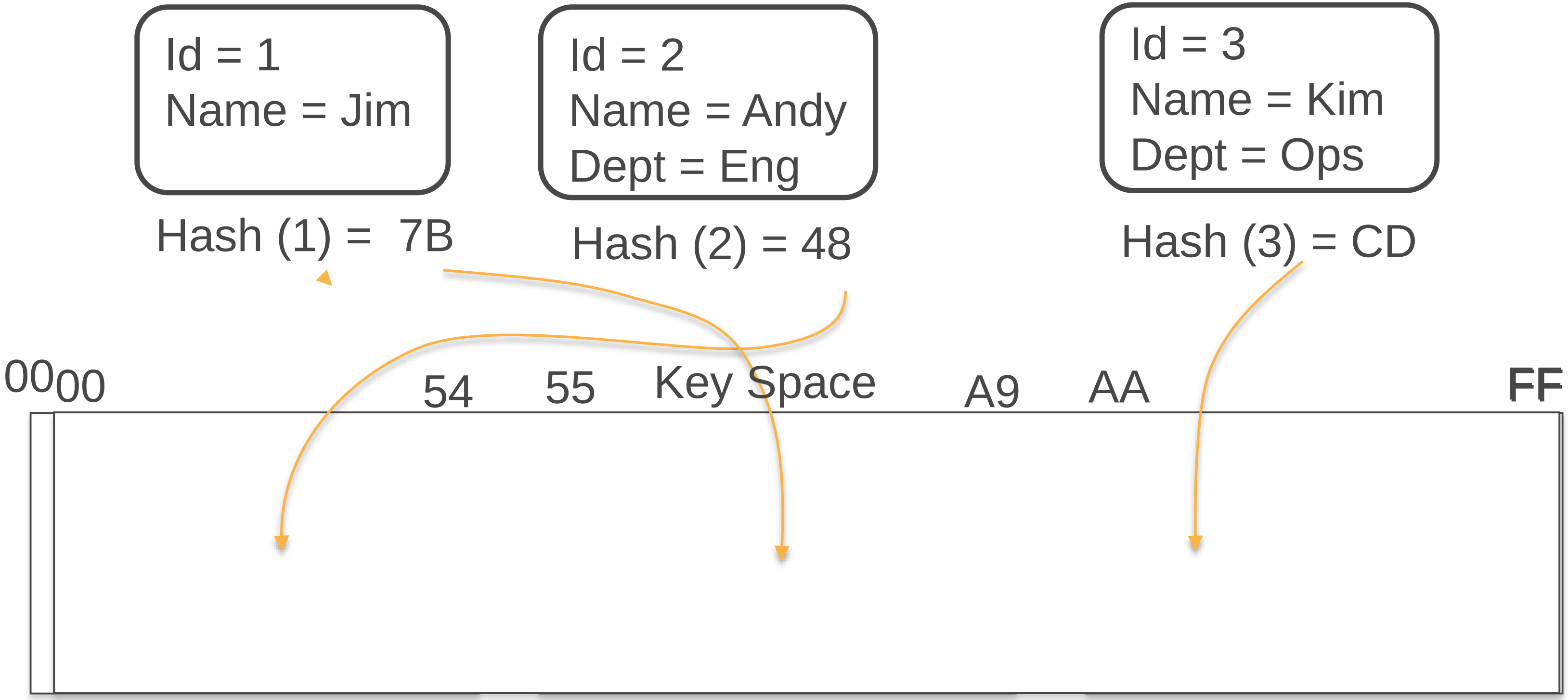
Mandatory
Key-value access pattern
Determines data distribution

Optional
Model 1:N relationships
Enables rich query capabilities

All items for key
==, <, >, >=, <=
“begins with”
“between”
“contains”
“in”
sorted results
counts
top/bottom N values

Partition Keys

Partition Key uniquely identifies an item
Partition Key is used for building an unordered hash index
Allows table to be partitioned for scale



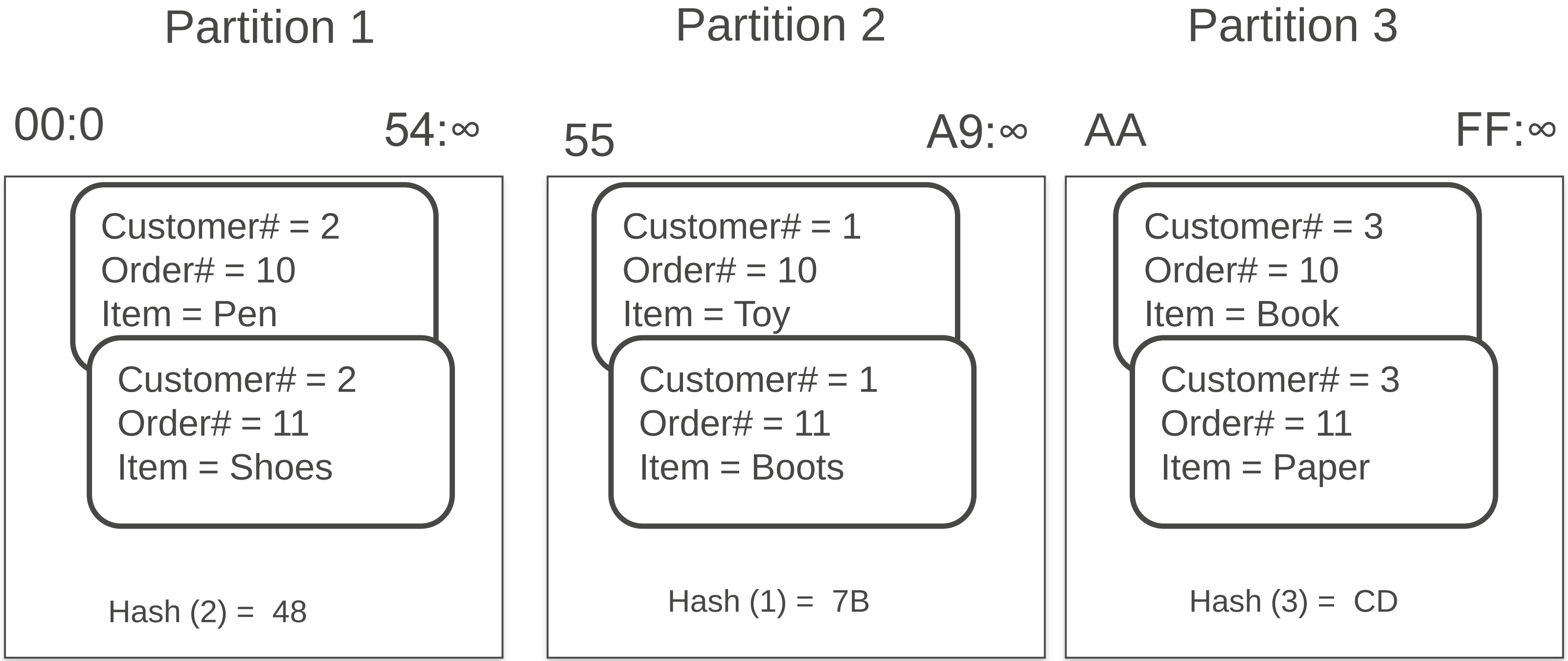
Partition:Sort Key

Partition:Sort Key uses two attributes together to uniquely identify an Item

Within unordered hash index, data is arranged by the sort key

No limit on the number of items (∞) per partition key

- Except if you have local secondary indexes



Partition 1

00:0

54:∞

Customer# = 2
Order# = 10
Item = Pen

Customer# = 2
Order# = 11
Item = Shoes

Hash (2) = 48

Partition 2

55

A9:∞

Customer# = 1
Order# = 10
Item = Toy

Customer# = 1
Order# = 11
Item = Boots

Hash (1) = 7B

Partition 3

AA

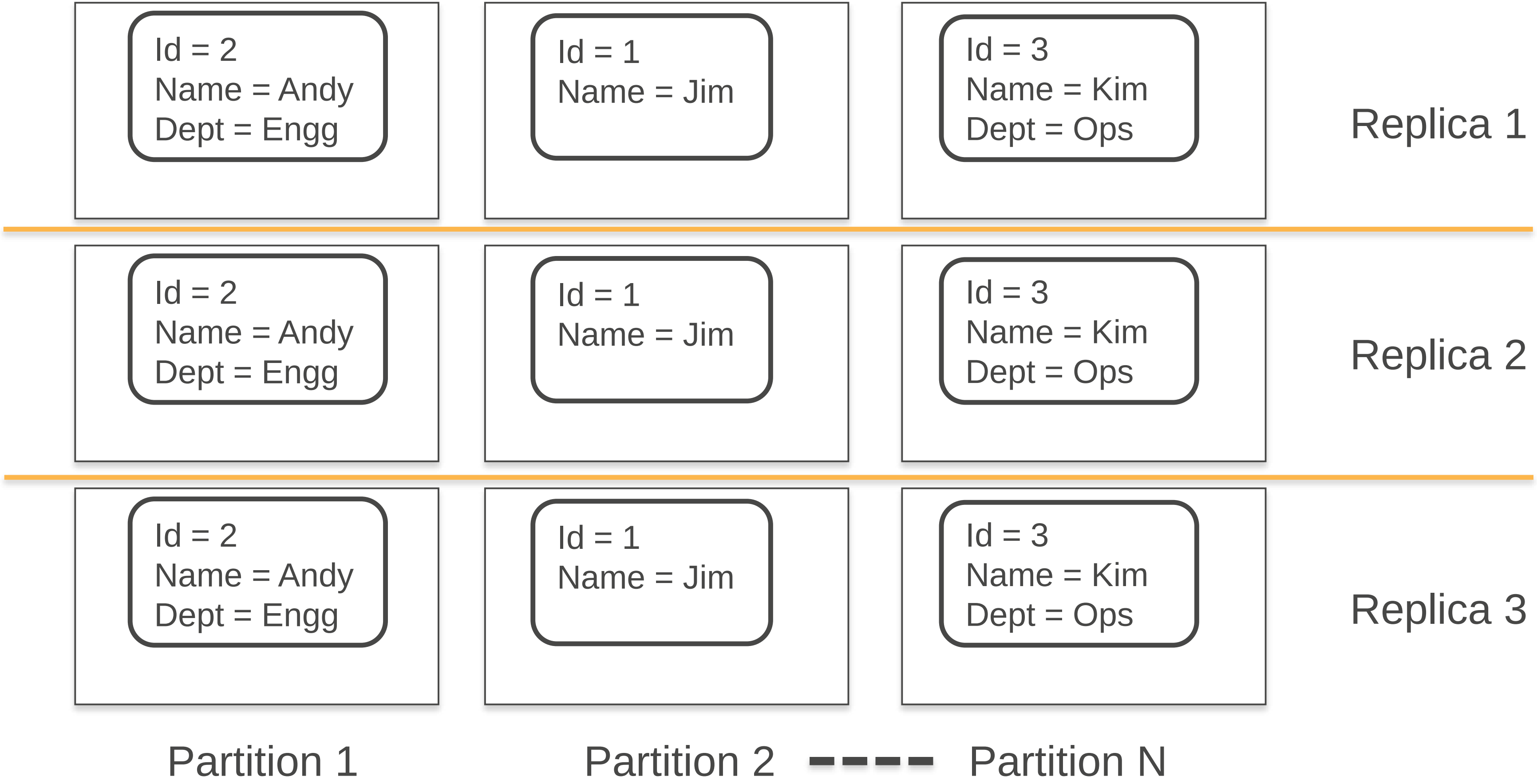
FF:∞

Customer# = 3
Order# = 10
Item = Book

Customer# = 3
Order# = 11
Item = Paper

Hash (3) = CD

Partitions are three-way replicated



Partition 1	Partition 2	Partition N	

Id = 2
Name = Andy
Dept = Engg
	Id = 1
	Name = Jim
		Id = 3
		Name = Kim
		Dept = Ops
			Replica 1

Id = 2
Name = Andy
Dept = Engg
	Id = 1
	Name = Jim
		Id = 3
		Name = Kim
		Dept = Ops
			Replica 2

Id = 2
Name = Andy
Dept = Engg
	Id = 1
	Name = Jim
		Id = 3
		Name = Kim
		Dept = Ops
			Replica 3

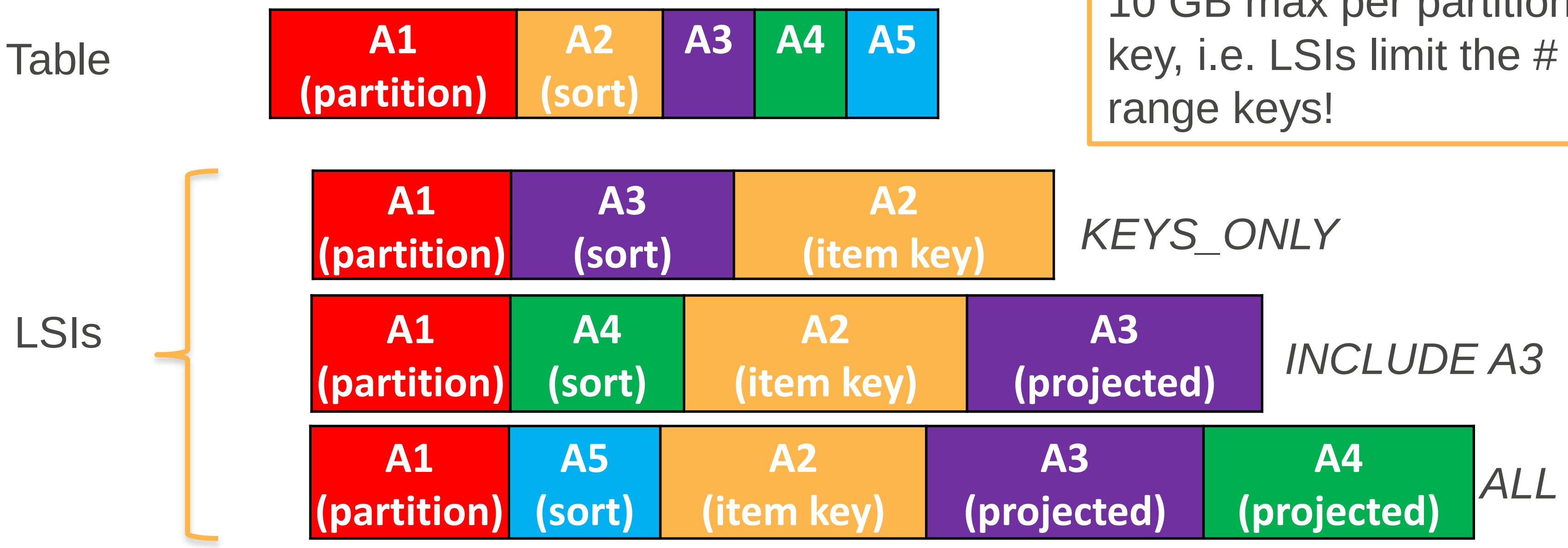
Indexes

Local secondary index (LSI)

Alternate sort key attribute

Index is local to a partition key

10 GB max per partition key, i.e. LSIs limit the # of range keys!



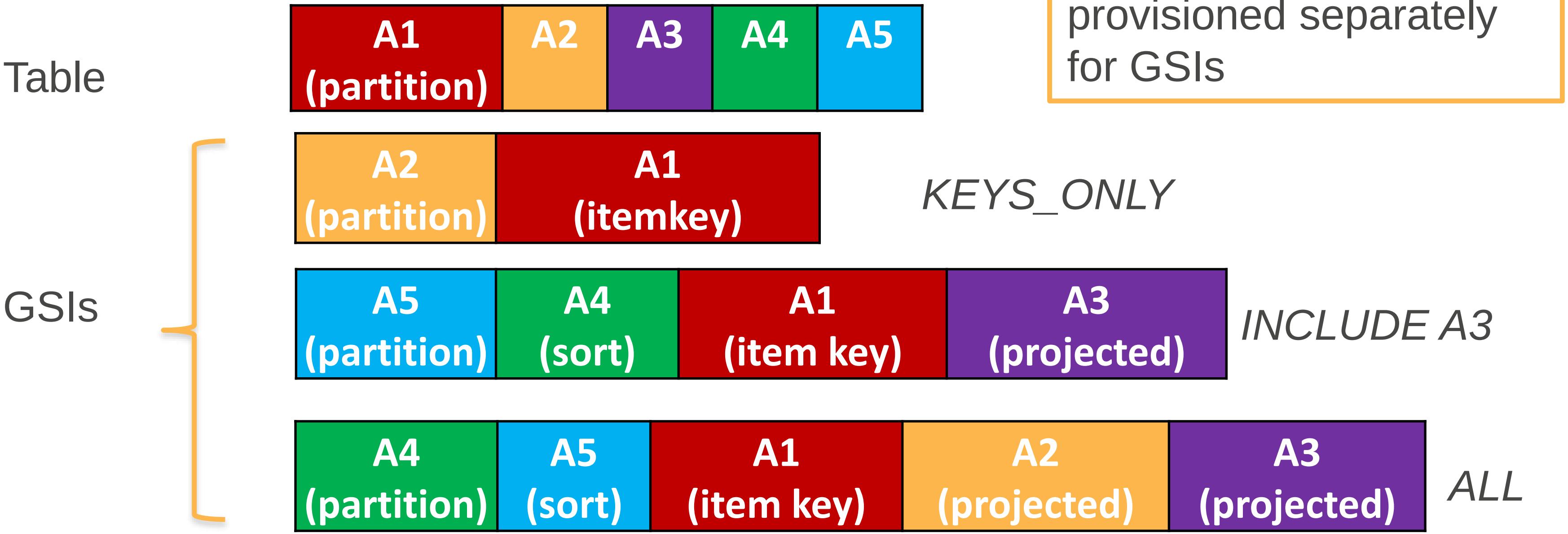
Global secondary index (GSI)

Online indexing

Alternate partition and/or sort key

Index is across all partition keys

RCUs/WCUs provisioned separately for GSIs



Table

A1 (partition) | A2 | A3 | A4 | A5

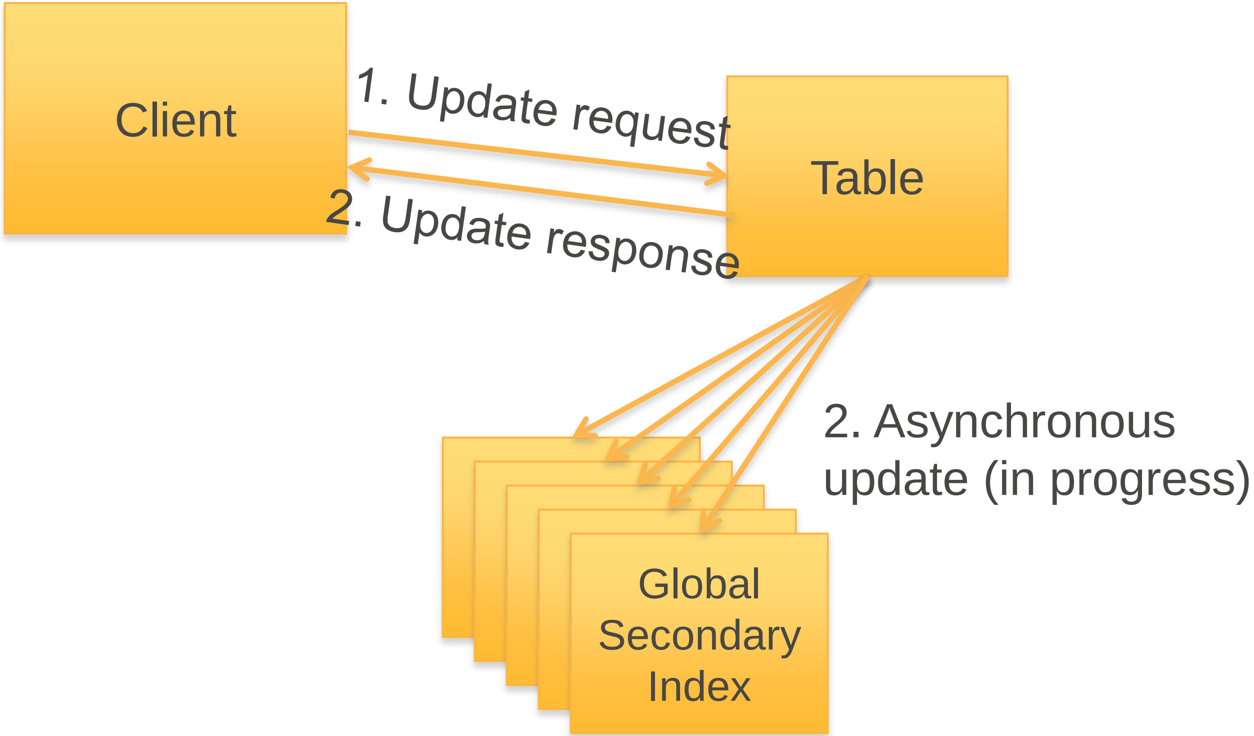
GSIs

A2 (partition) | A1 (itemkey) — *KEYS_ONLY*

A5 (partition) | A4 (sort) | A1 (item key) | A3 (projected) — *INCLUDE A3*

A4 (partition) | A5 (sort) | A1 (item key) | A2 (projected) | A3 (projected) — *ALL*

How do GSI updates work?



If GSIs don’t have enough write capacity, table writes will be throttled!

LSI or GSI?

LSI can be modeled as a GSI

If data size in an item collection > 10 GB, use GSI

If eventual consistency is okay for your scenario, use GSI!

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 range keys 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

RCU

WCU

Partitioning math

Number of Partitions	
By Capacity	(Total RCU / 3000) + (Total WCU / 1000)
By Size	Total Size / 10 GB
Total Partitions	CEILING(MAX (Capacity, Size))

In the future, these details might change…

Partitioning example

Table size = 8 GB, RCUs = 5000, WCUs = 500

Number of Partitions	
By Capacity	(5000 / 3000) + (500 / 1000) = 2.17
By Size	8 / 10 = 0.8
Total Partitions	CEILING(MAX (2.17, 0.8)) = 3

RCUs and WCUs are uniformly spread across partitions

RCUs per partition = 5000/3 = 1666.67
WCUs per partition = 500/3 = 166.67
Data/partition = 10/3 = 3.33 GB

What causes throttling?

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

Non-uniform workloads

- Hot keys/hot partitions
- Very large bursts

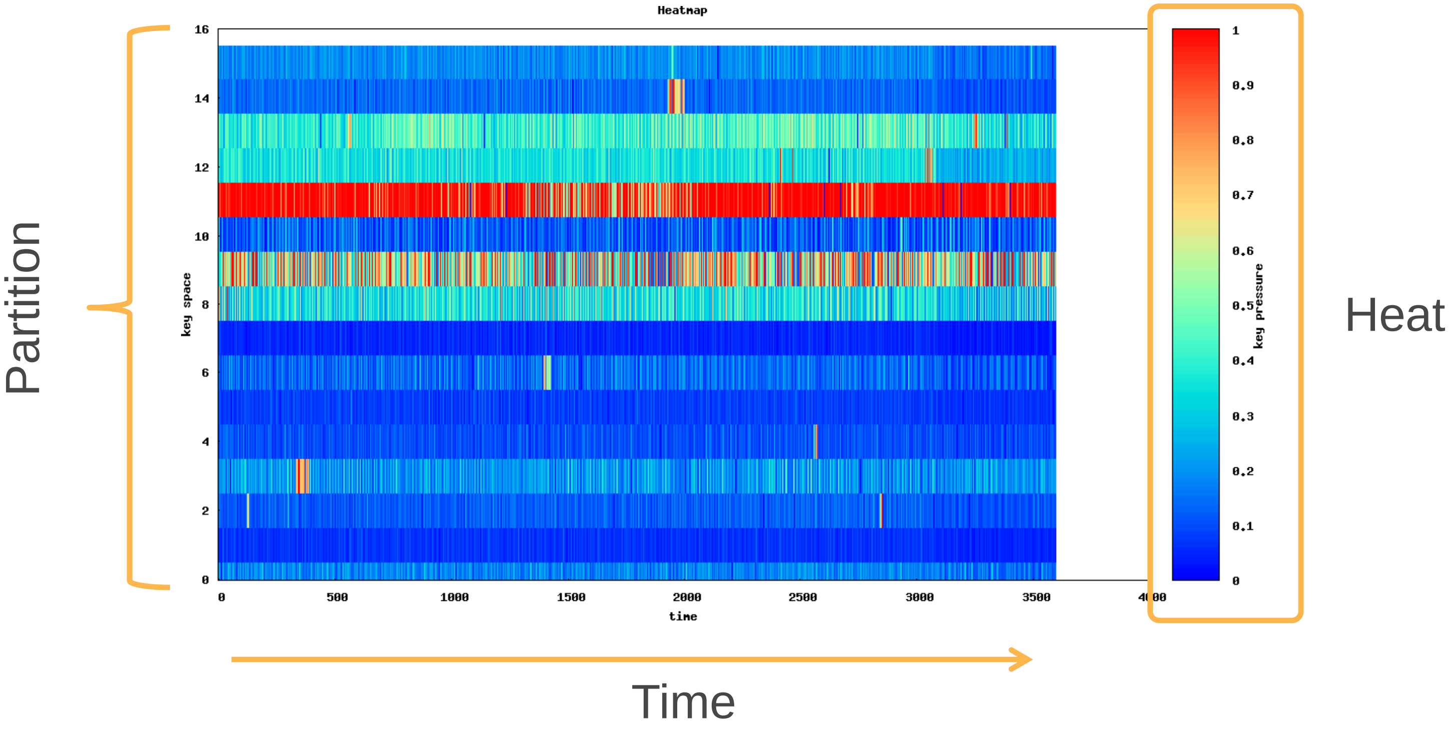
Mixing hot data with cold data

- Use a table per time period

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 bad NoSQL looks like…



Partition

Heat

Time

Getting the most out of DynamoDB throughput

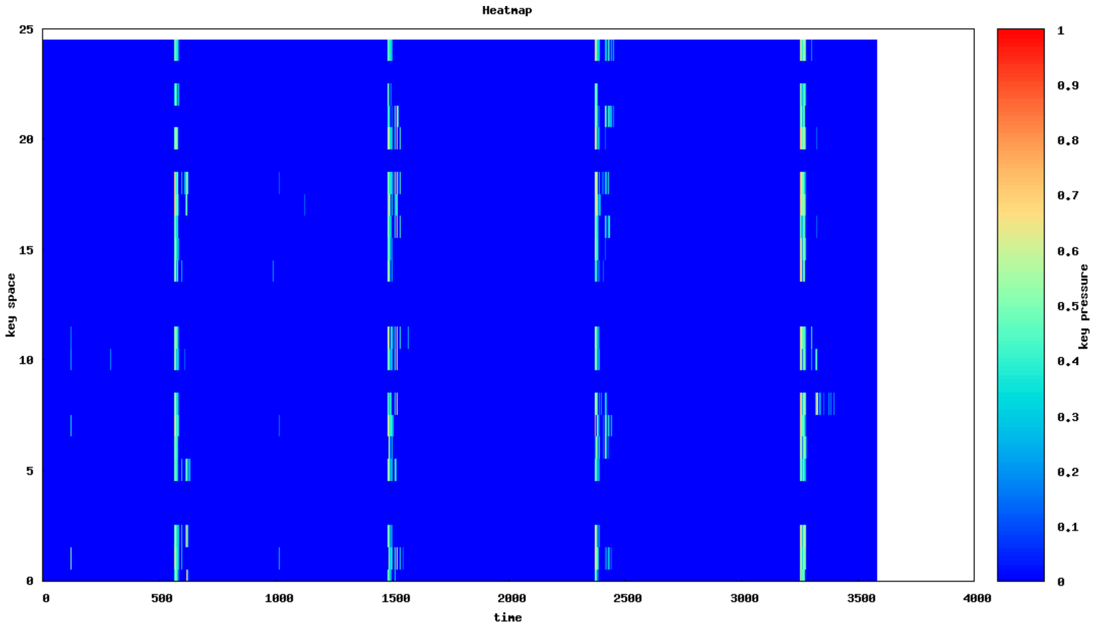
“To get the most out of DynamoDB throughput, create tables where the hash key element has a large number of distinct values, and values are requested fairly uniformly, as randomly as possible.”

—*DynamoDB Developer Guide*

Space: access is evenly spread over the key-space

Time: requests arrive evenly spaced in time

Much better picture…



Data modeling

1:1 relationships or key-values

Use a table or GSI with an alternate partition key

Use GetItem or BatchGetItem API

Example: Given an SSN or license number, get attributes

Users Table	
Partiton key	Attributes
SSN = 123-45-6789	Email = johndoe@nowhere.com, License = TDL25478134
SSN = 987-65-4321	Email = maryfowler@somewhere.com, License = TDL78309234

Users-Email-GSI	
Partition key	Attributes
License = TDL78309234	Email = maryfowler@somewhere.com, SSN = 987-65-4321
License = TDL25478134	Email = johndoe@nowhere.com, SSN = 123-45-6789

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		
Partition 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

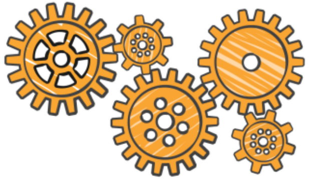
Use Query API

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

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

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

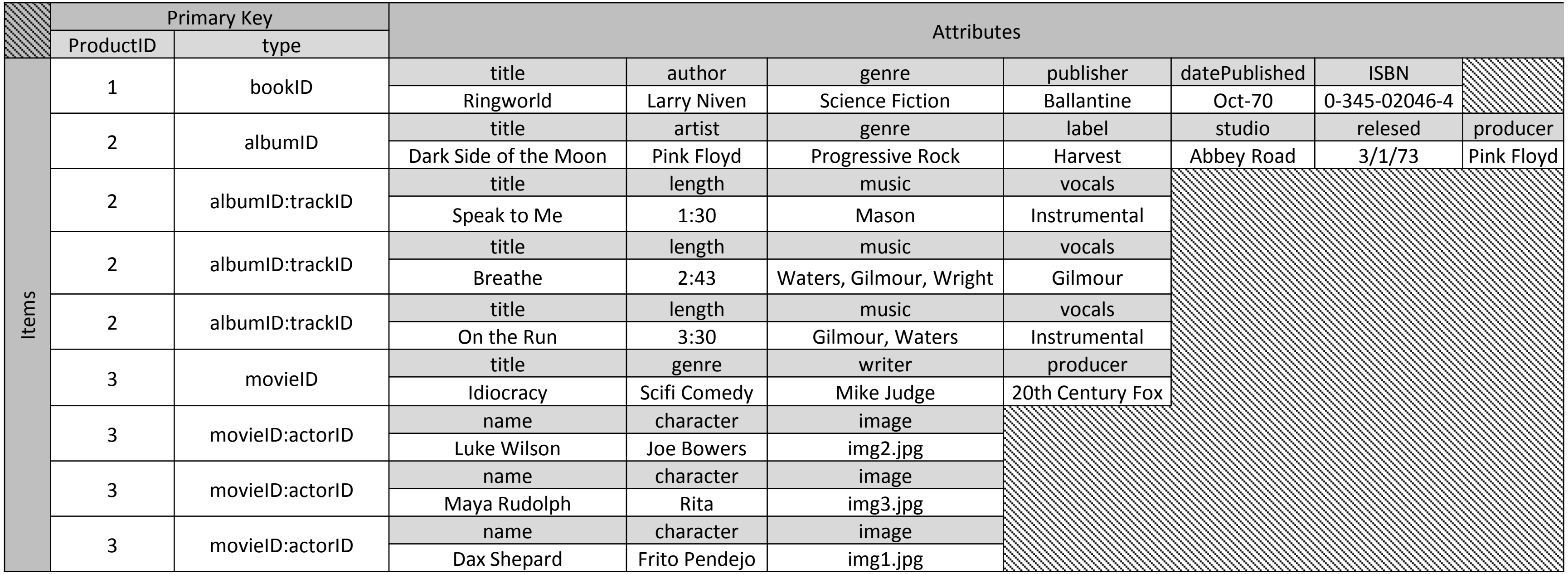
Hierarchical Data



Tiered relational data structures

Hierarchical Data Structures as Items…

Use composite sort key to define a Hierarchy
Highly selective result sets with sort queries
Index anything, scales to any size



	Primary Key		Attributes						
	ProductID	type							
Items	1	bookID	title	author	genre	publisher	datePublished	ISBN	
			Ringworld	Larry Niven	Science Fiction	Ballantine	Oct-70	0-345-02046-4	
	2	albumID	title	artist	genre	label	studio	relesed	producer
			Dark Side of the Moon	Pink Floyd	Progressive Rock	Harvest	Abbey Road	3/1/73	Pink Floyd
	2	albumID:trackID	title	length	music	vocals			
			Speak to Me	1:30	Mason	Instrumental			
	2	albumID:trackID	title	length	music	vocals			
			Breathe	2:43	Waters, Gilmour, Wright	Gilmour			
	2	albumID:trackID	title	length	music	vocals			
			On the Run	3:30	Gilmour, Waters	Instrumental			
	3	movieID	title	genre	writer	producer			
			Idiocracy	Scifi Comedy	Mike Judge	20th Century Fox			
	3	movieID:actorID	name	character	image				
			Luke Wilson	Joe Bowers	img2.jpg				
	3	movieID:actorID	name	character	image				
			Maya Rudolph	Rita	img3.jpg				
	3	movieID:actorID	name	character	image				
			Dax Shepard	Frito Pendejo	img1.jpg				

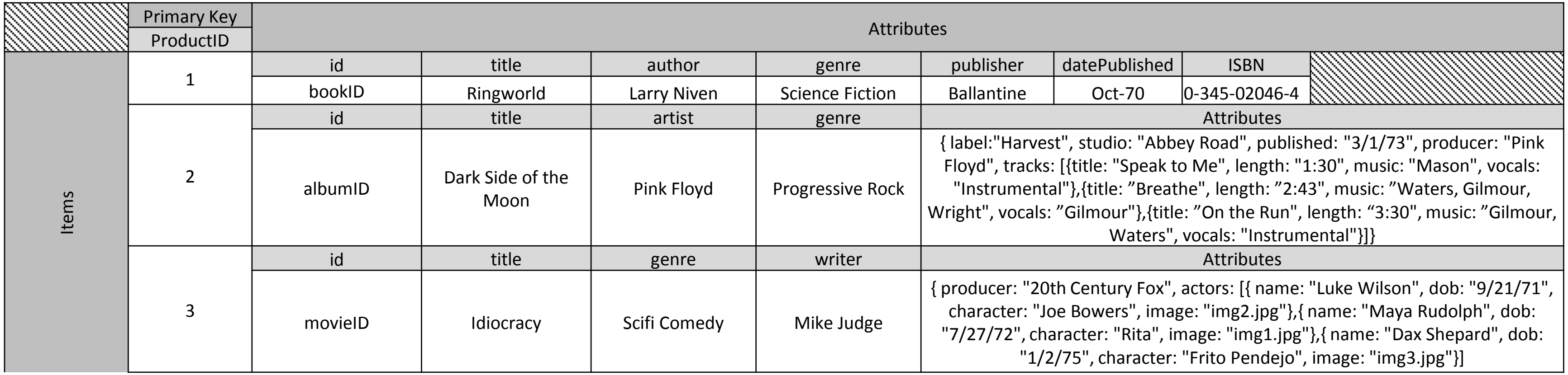
… or as Documents (JSON)

JSON data types (M, L, BOOL, NULL)

Document SDKs Available

Indexing only via Streams/Lambda

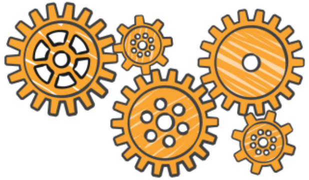
400KB max item size (limits hierarchical data structure)



	Primary Key	Attributes							
	ProductID								
Items	1	id	title	author	genre	publisher	datePublished	ISBN	
		bookID	Ringworld	Larry Niven	Science Fiction	Ballantine	Oct-70	0-345-02046-4	
	2	id	title	artist	genre	Attributes			
		albumID	Dark Side of the Moon	Pink Floyd	Progressive Rock	{ label:"Harvest", studio: "Abbey Road", published: "3/1/73", producer: "Pink Floyd", tracks: [{title: "Speak to Me", length: "1:30", music: "Mason", vocals: "Instrumental"},{title: ”Breathe", length: ”2:43", music: ”Waters, Gilmour, Wright", vocals: ”Gilmour"},{title: ”On the Run", length: “3:30", music: ”Gilmour, Waters", vocals: "Instrumental"}]}			
	3	id	title	genre	writer	Attributes			
		movieID	Idiocracy	Scifi Comedy	Mike Judge	{ producer: "20th Century Fox", actors: [{ name: "Luke Wilson", dob: "9/21/71", character: "Joe Bowers", image: "img2.jpg"},{ name: "Maya Rudolph", dob: "7/27/72", character: "Rita", image: "img1.jpg"},{ name: "Dax Shepard", dob: "1/2/75", character: "Frito Pendejo", image: "img3.jpg"}]			

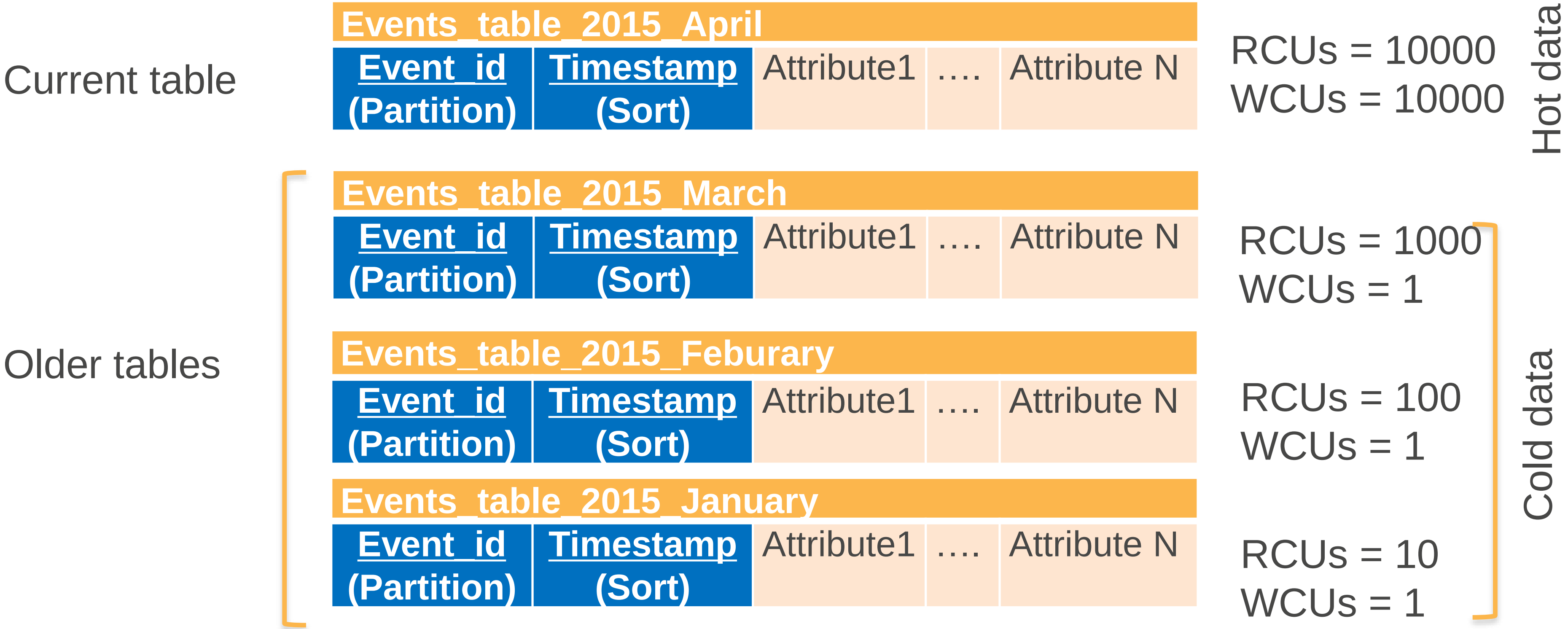
Scenarios and best practices

Event logging



Storing time series data

Time series tables



Current table

Events_table_2015_April

Event_id (Partition)	Timestamp (Sort)	Attribute1	….	Attribute N

RCUs = 10000
WCUs = 10000

Hot data

Older tables

Events_table_2015_March

Event_id (Partition)	Timestamp (Sort)	Attribute1	….	Attribute N

RCUs = 1000
WCUs = 1

Events_table_2015_Feburary

Event_id (Partition)	Timestamp (Sort)	Attribute1	….	Attribute N

RCUs = 100
WCUs = 1

Events_table_2015_January

Event_id (Partition)	Timestamp (Sort)	Attribute1	….	Attribute N

RCUs = 10
WCUs = 1

Cold data

Don’t mix hot and cold data; archive cold data to Amazon S3

☑ Use a table per time period

Pre-create daily, weekly, monthly tables

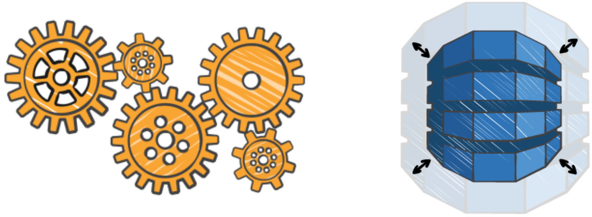
Provision required throughput for current table

Writes go to the current table

Turn off (or reduce) throughput for older tables

Important when: Dealing with time series data

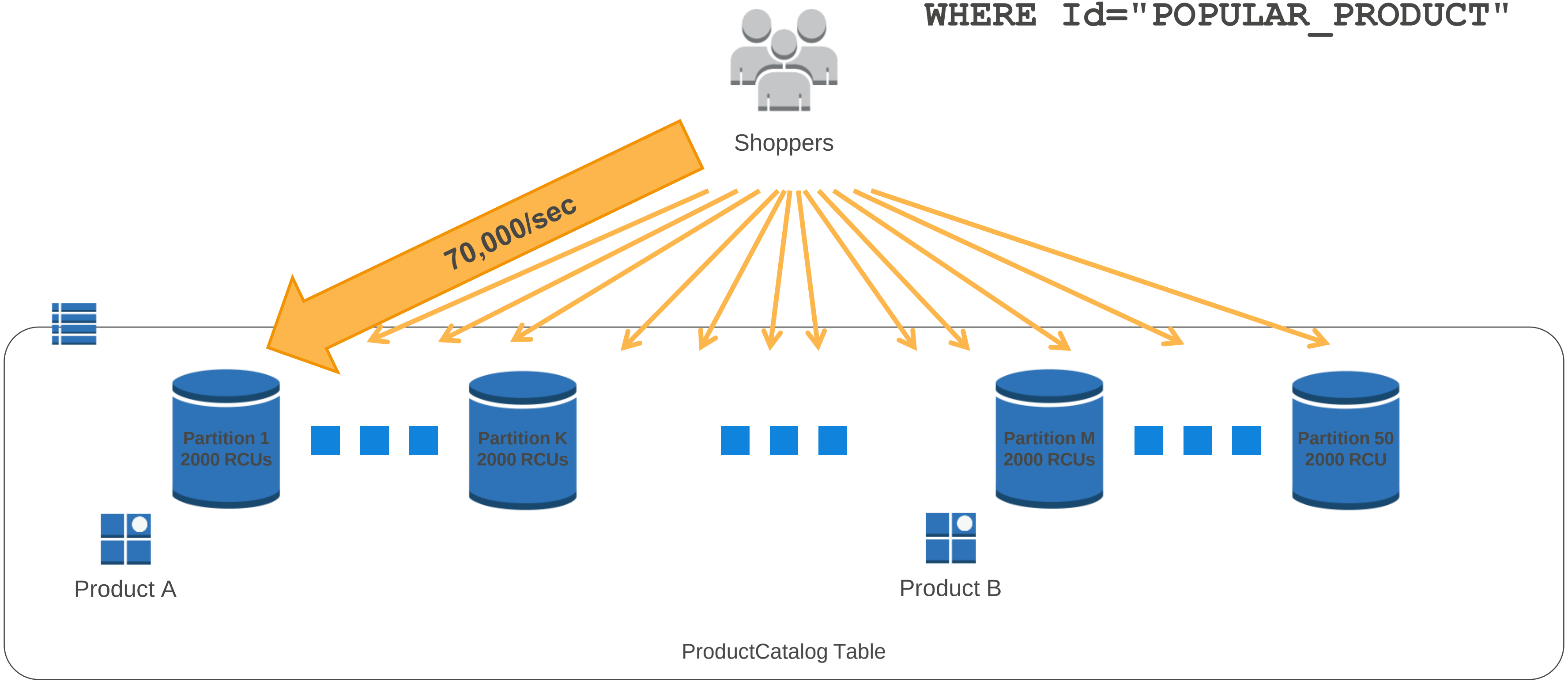
Product catalog



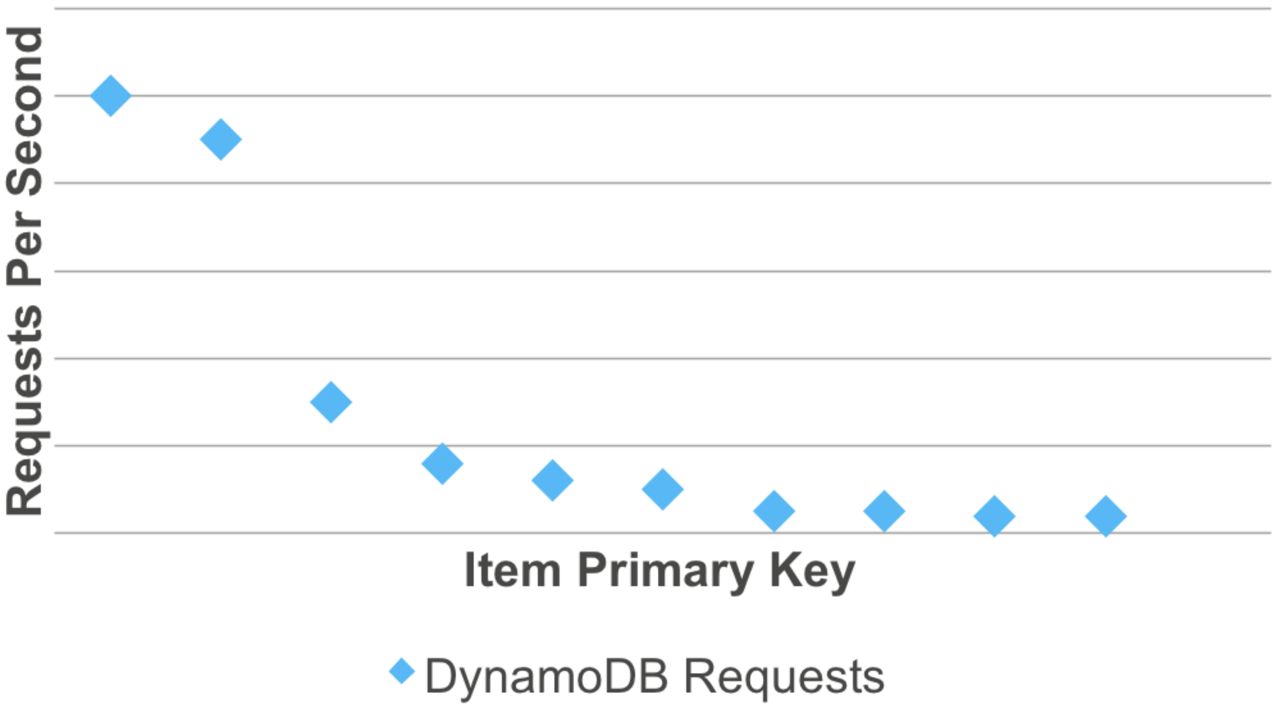
Popular items (read)

Scaling bottlenecks

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

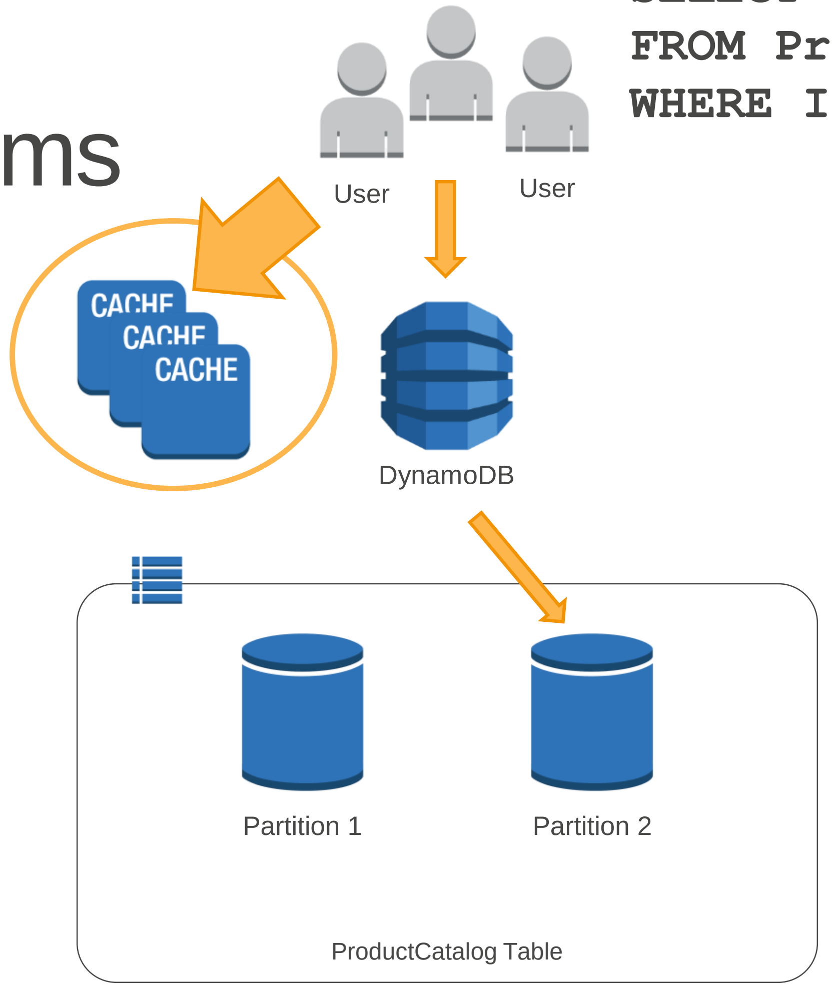


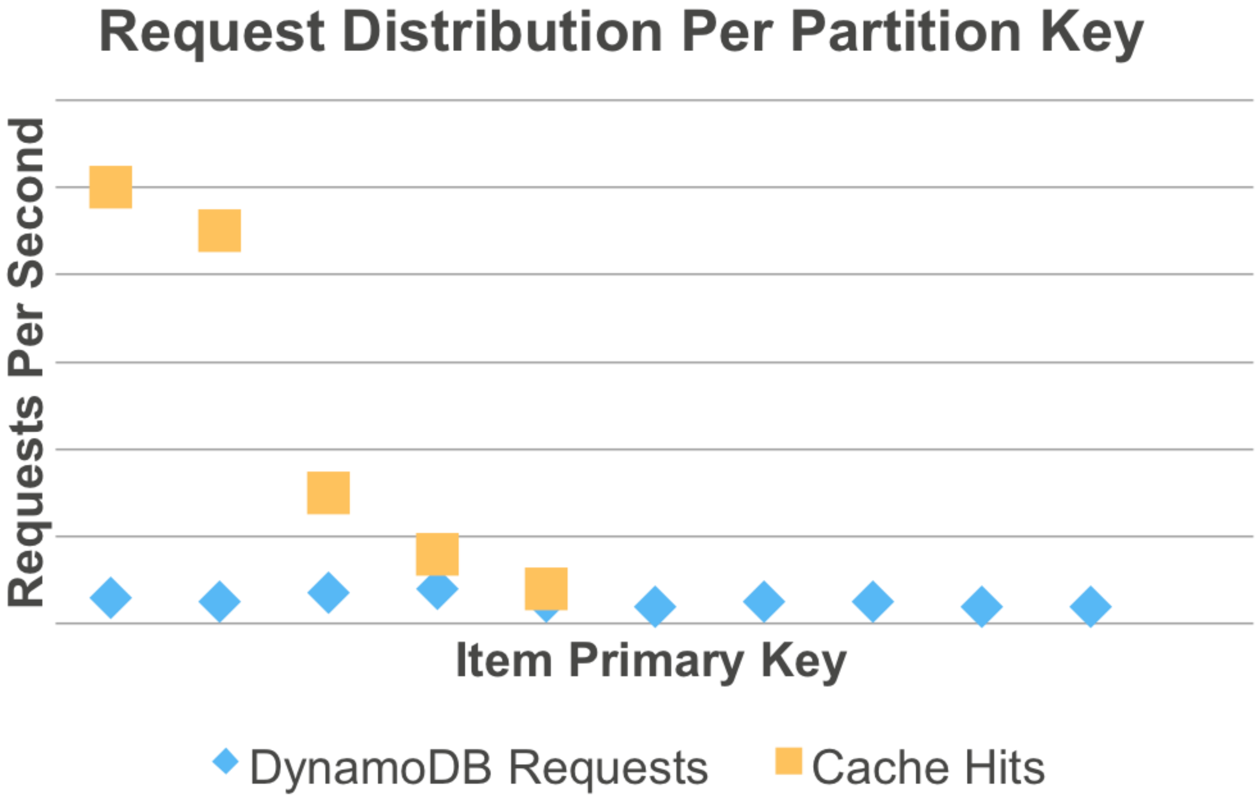
Request Distribution Per Partition Key



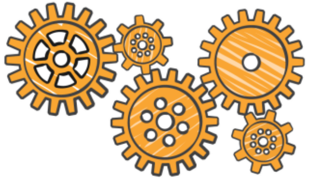
Cache popular items

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

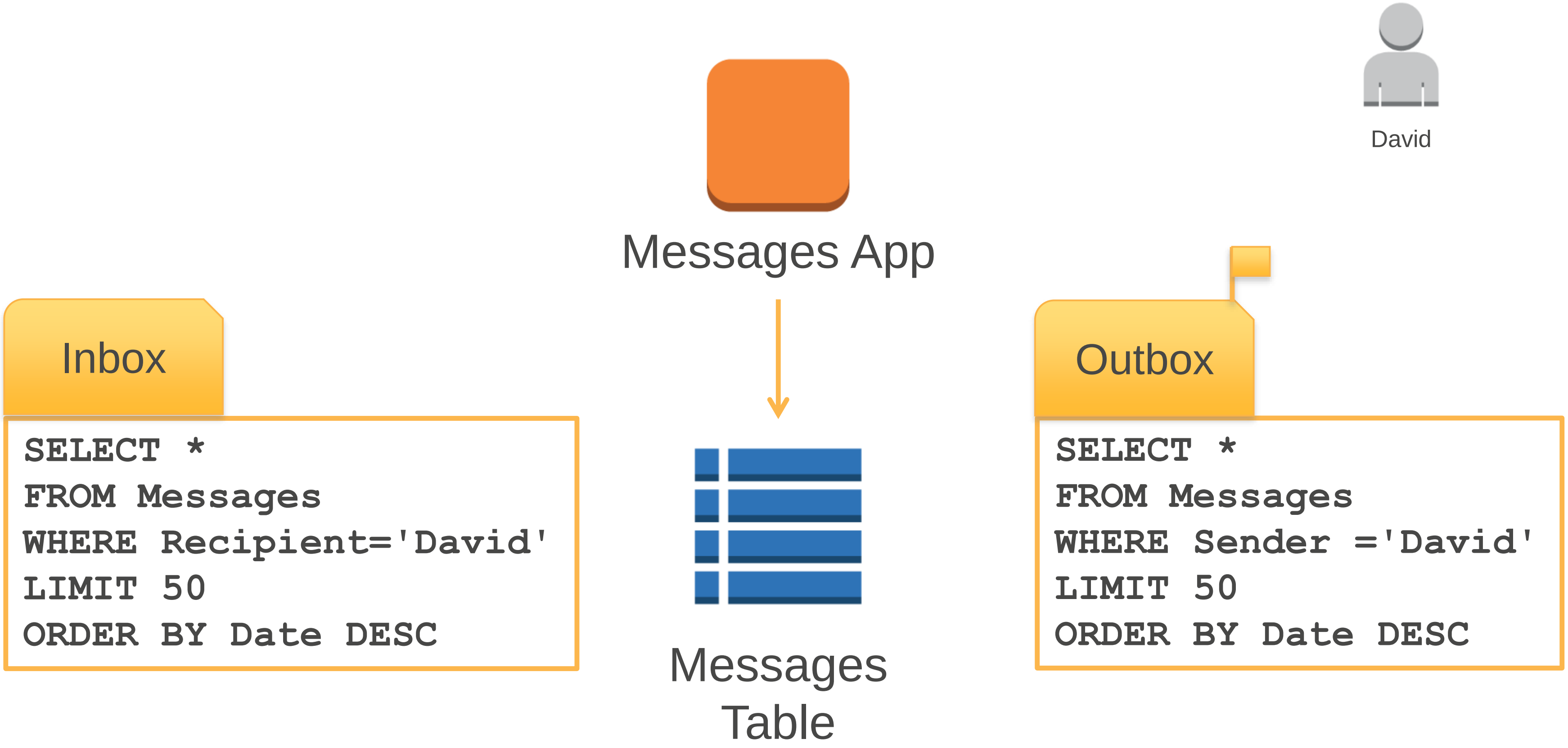



Request Distribution Per Partition Key
Requests Per Second
Item Primary Key
◆ DynamoDB Requests
■ Cache Hits

Messaging app



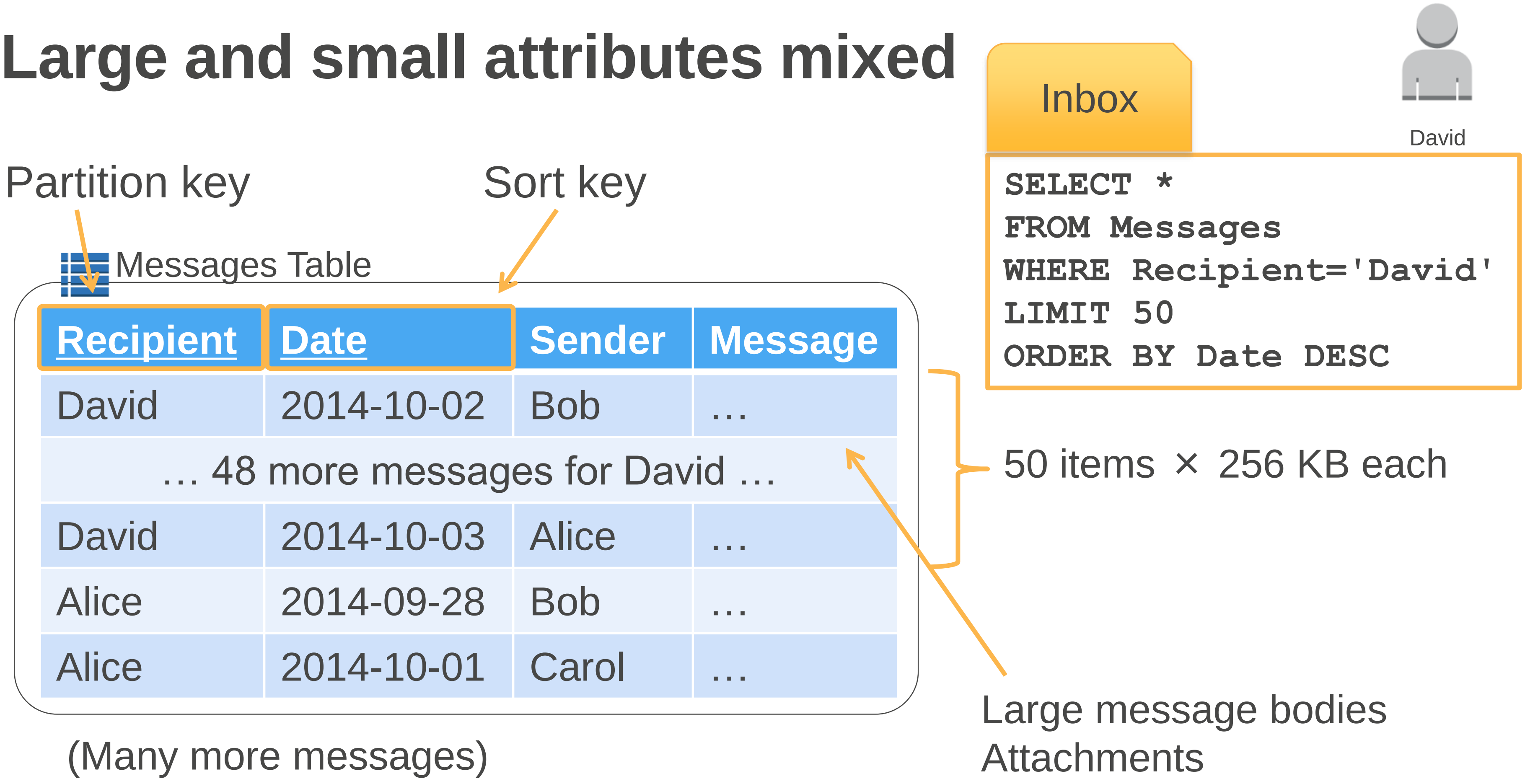
Large items
Filters vs. indexes
M:N Modeling—inbox and outbox



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

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

Large and small attributes mixed



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)

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

50 items × 256 KB each

Large message bodies
Attachments

Computing inbox query cost

$$\underset{\substack{\uparrow \\ \text{Items evaluated by query}}}{50} \quad * \quad \underset{\substack{\uparrow \\ \text{Average item size}}}{256\text{KB}} \quad * \quad \underset{\substack{\uparrow \\ \text{Conversion ratio}}}{(1\ \text{RCU} / 4\text{KB})} \quad * \quad \underset{\substack{\uparrow \\ \text{Eventually consistent reads}}}{(1 / 2)} \quad = \quad \boxed{1600\ \text{RCU}}$$

Separate the bulk data

Uniformly distributes large item reads

David

(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

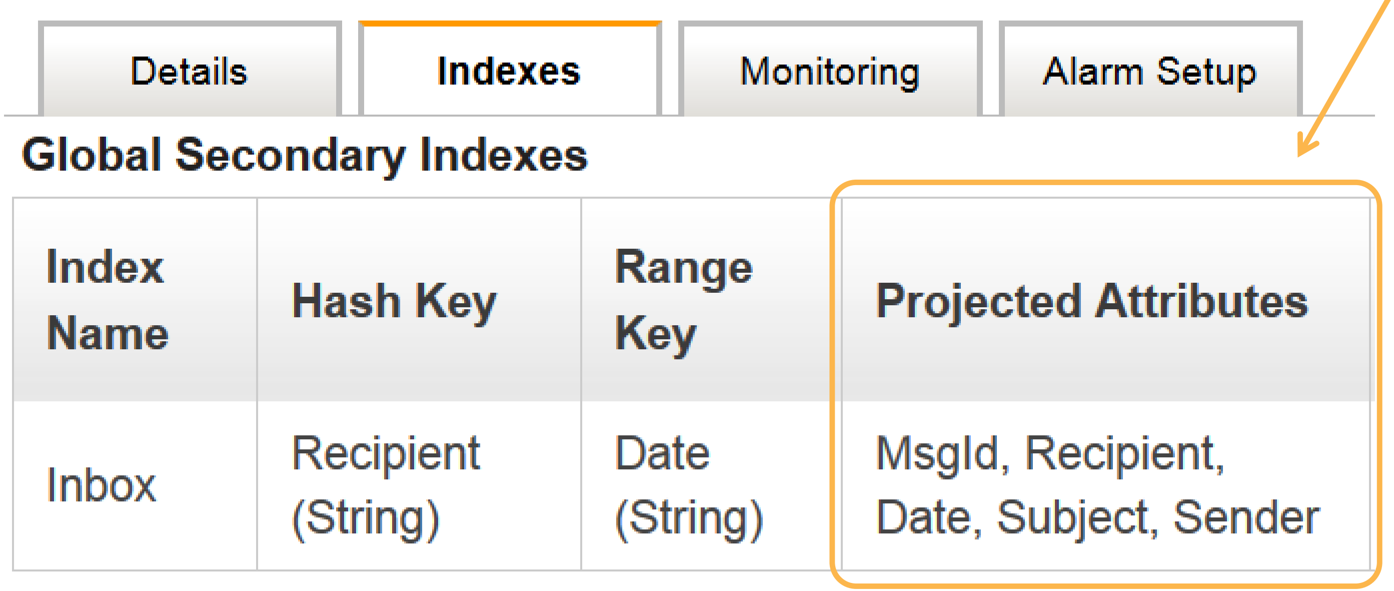
Recipient	Date	Sender	Subject	MsgId
David	2014-10-02	Bob	Hi!…	afed
David	2014-10-03	Alice	RE: The…	3kf8
Alice	2014-09-28	Bob	FW: Ok…	9d2b
Alice	2014-10-01	Carol	Hi!...	ct7r

Messages Table

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

Inbox GSI

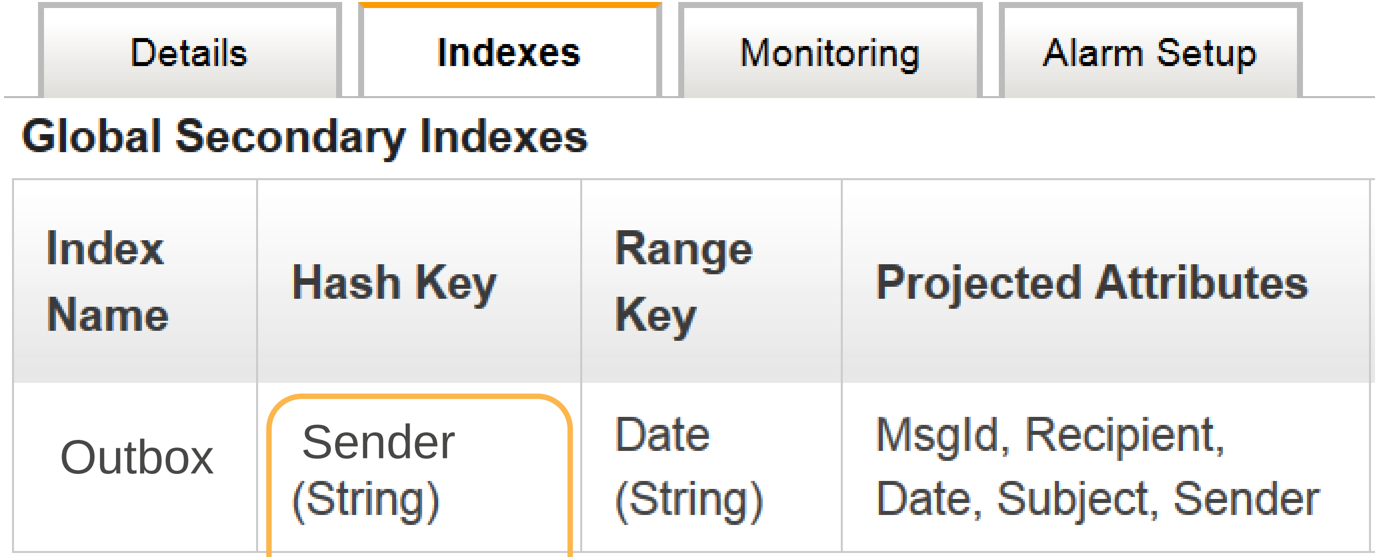
Define which attributes to copy into the index



Global Secondary Indexes

Index Name	Hash Key	Range Key	Projected Attributes
Inbox	Recipient (String)	Date (String)	MsgId, Recipient, Date, Subject, Sender

Outbox GSI

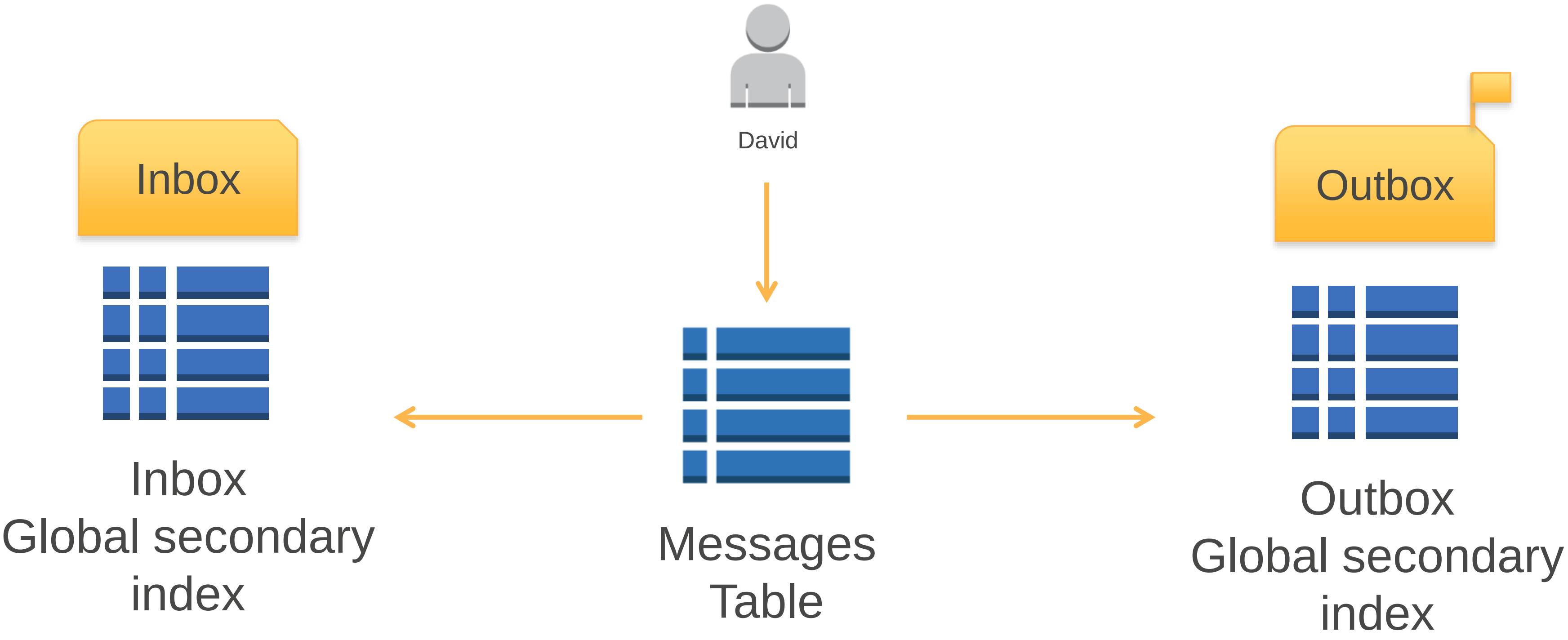


Global Secondary Indexes

Index Name	Hash Key	Range Key	Projected Attributes
Outbox	Sender (String)	Date (String)	MsgId, Recipient, Date, Subject, Sender

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

Messaging app



David

Inbox

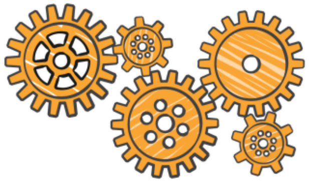
Inbox
Global secondary
index

Messages
Table

Outbox

Outbox
Global secondary
index

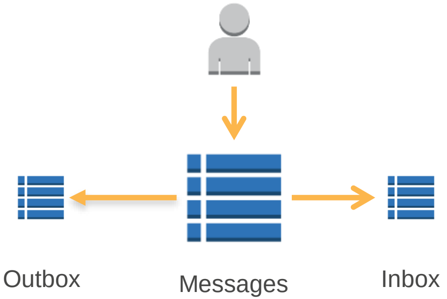
☑ Distribute large items



Reduce one-to-many item sizes

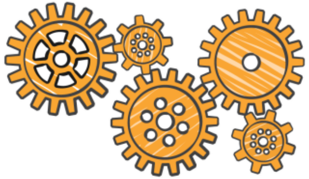
Configure secondary index projections

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

Hierarchical Data Structures

Partition key

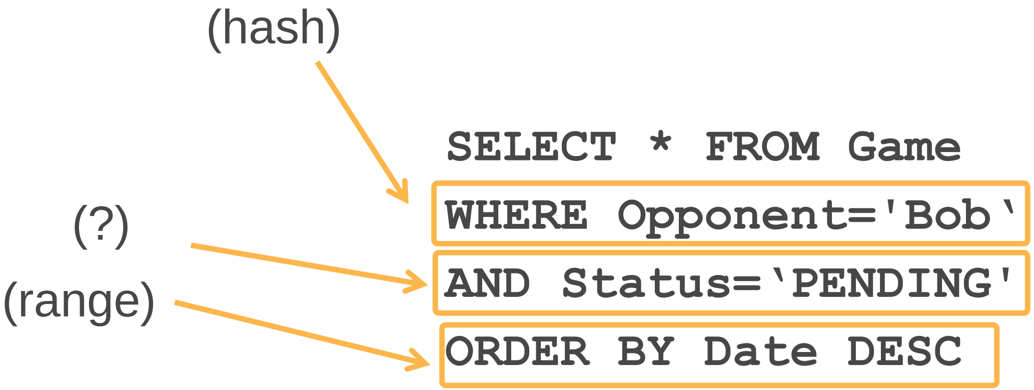
Games Table

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

Query for incoming game requests

DynamoDB indexes provide partiton and sort

What about queries for two equalities and a sort?



(hash)

(?)

(range)

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

Approach 1: Query filter

Bob

Partition key

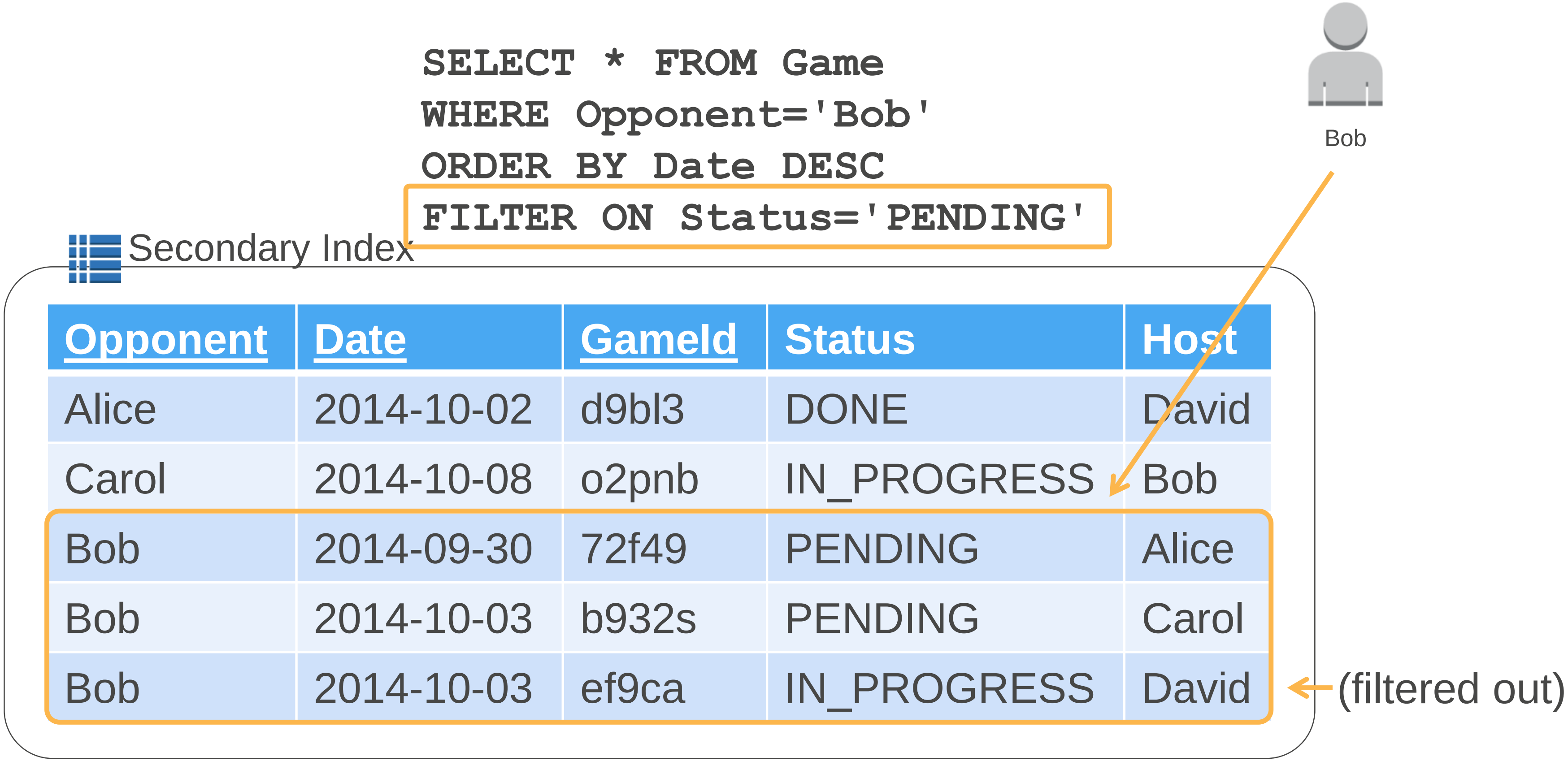
Sort key

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'
```



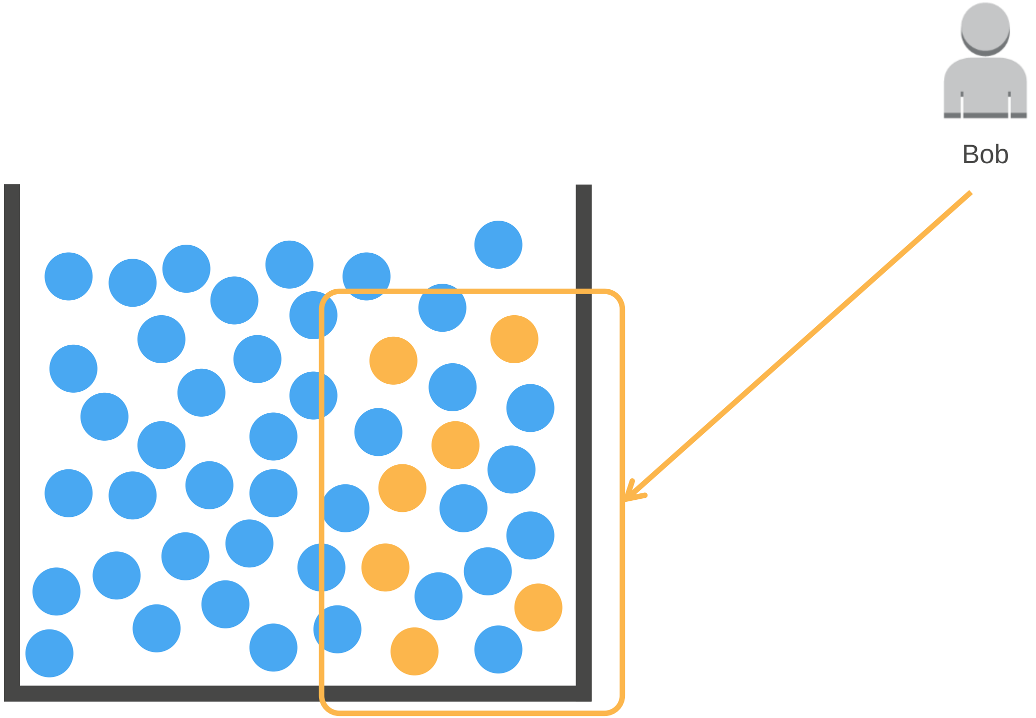
Bob

Secondary Index

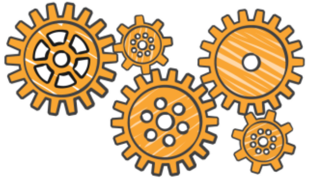
Opponent	Date	GameId	Status	Host
Alice	2014-10-02	d9bl3	DONE	David
Carol	2014-10-08	o2pnb	IN_PROGRESS	Bob
Bob	2014-09-30	72f49	PENDING	Alice
Bob	2014-10-03	b932s	PENDING	Carol
Bob	2014-10-03	ef9ca	IN_PROGRESS	David

(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

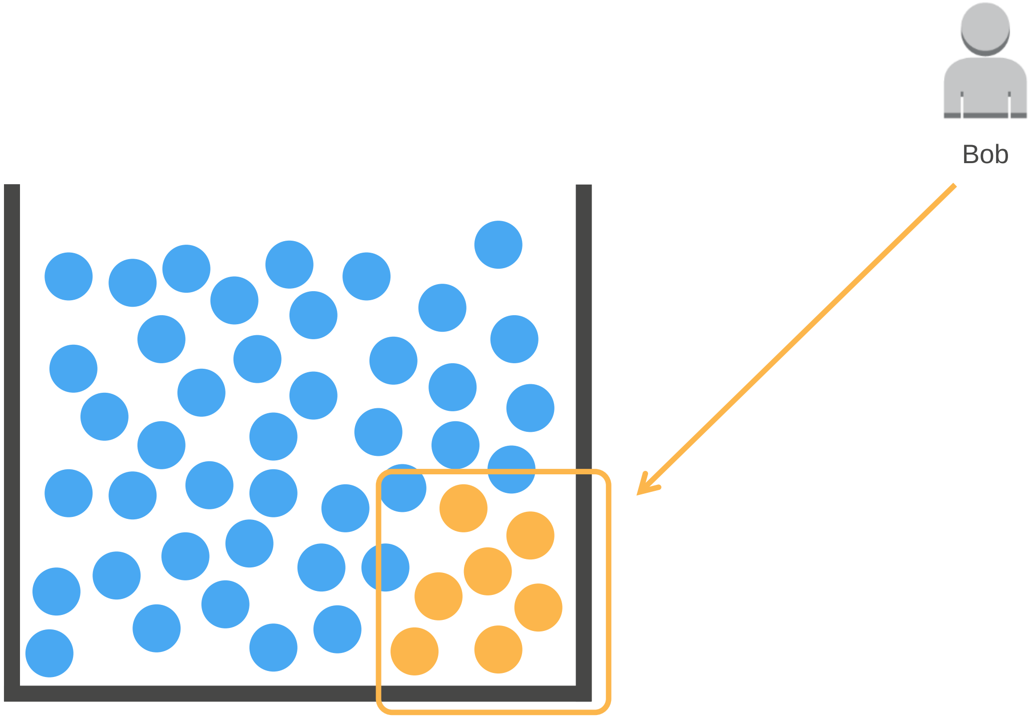
```
SELECT * FROM Game
WHERE Opponent='Bob'
   AND StatusDate BEGINS_WITH 'PENDING'
```

Bob

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

Game-scores-table

Id (Partition)	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	

Scan sparse GSIs

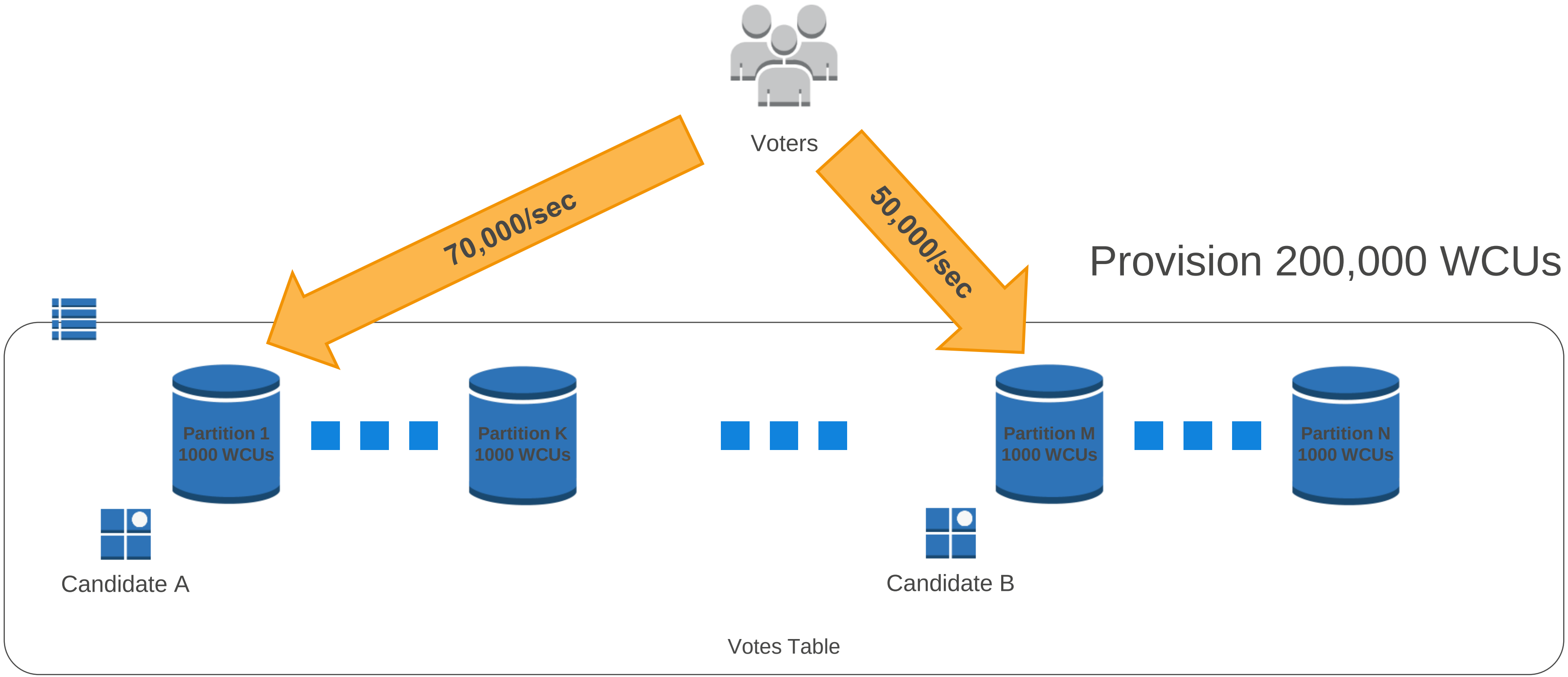
Award-GSI

Award (Partition)	Id	User	Score
Champ	4	Mary	2000

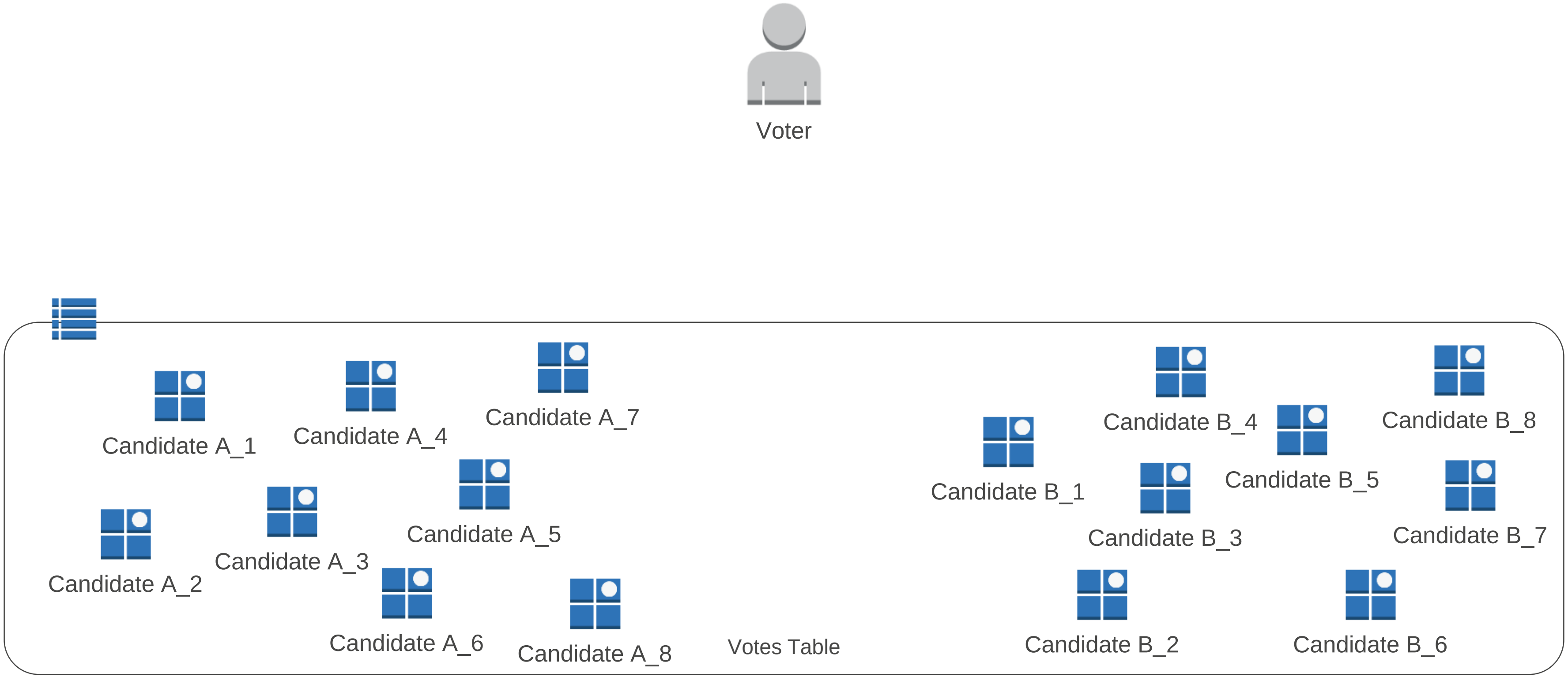
Real-Time voting

Write-heavy items

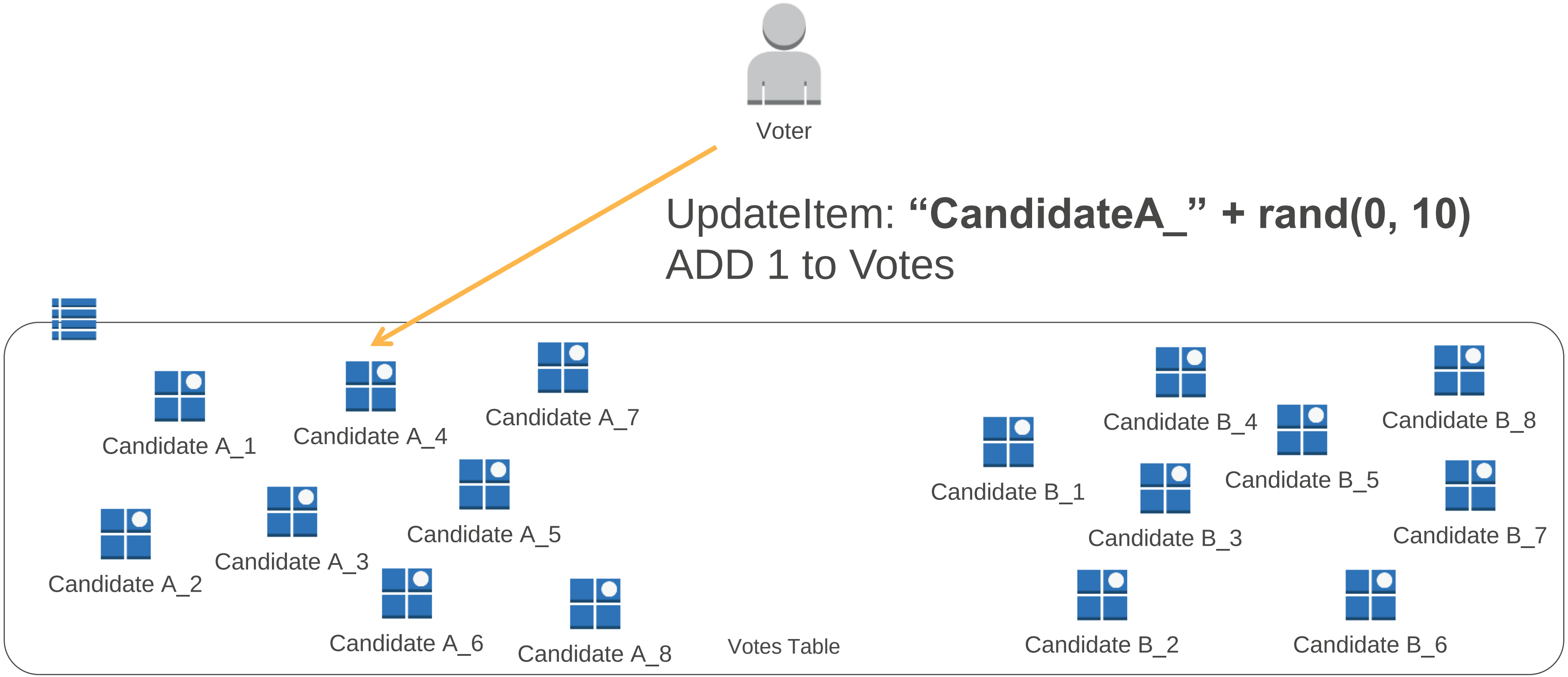
Scaling bottlenecks



Write sharding

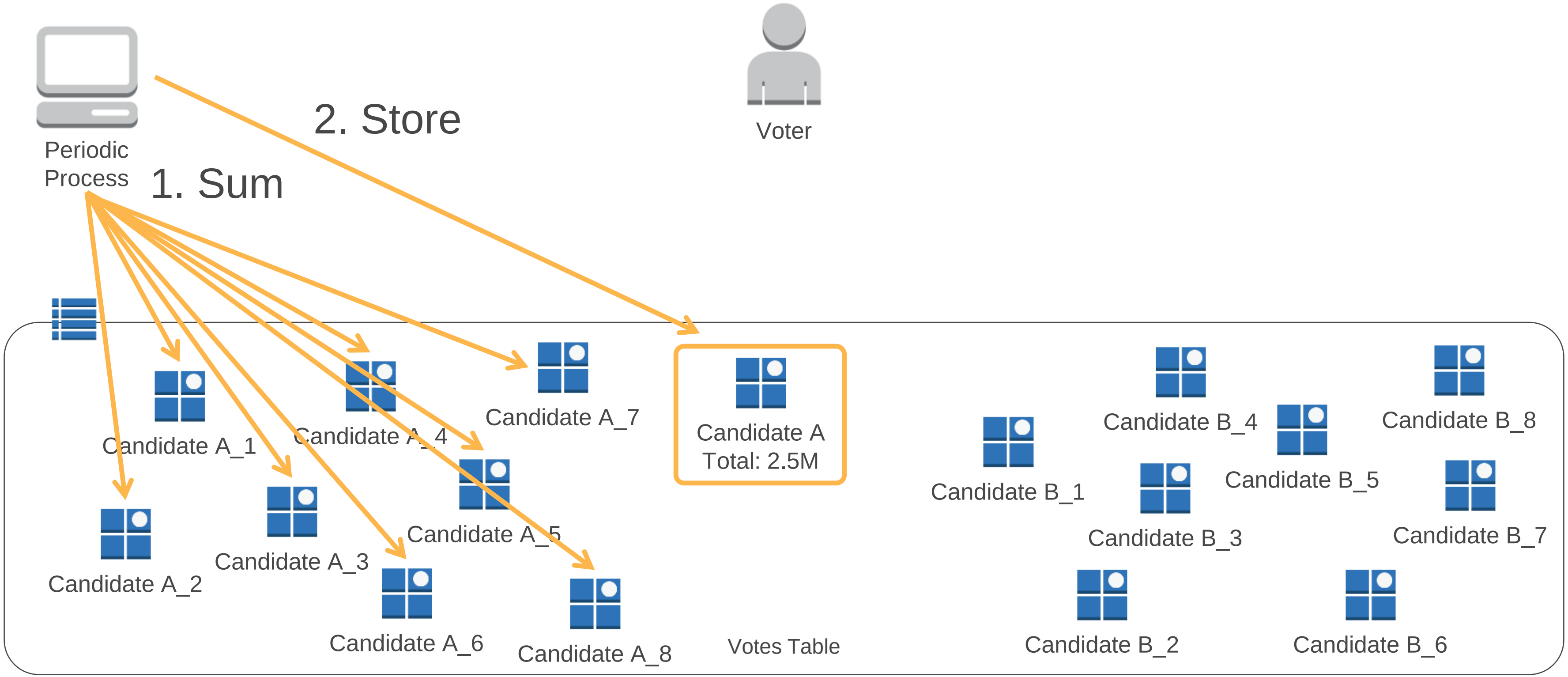


Write sharding



UpdateItem: **“CandidateA_” + rand(0, 10)**
ADD 1 to Votes

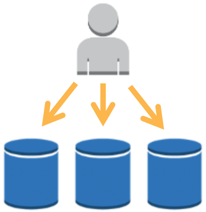
Shard aggregation



☑ Shard write-heavy partition keys

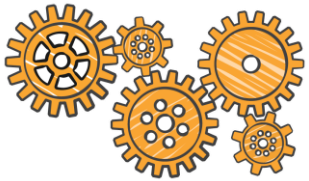
Trade off read cost for write scalability

Consider throughput per partition key



Important when: Your write workload is not horizontally scalable

☑ Replace filter with indexes



Concatenate attributes to form useful secondary index keys

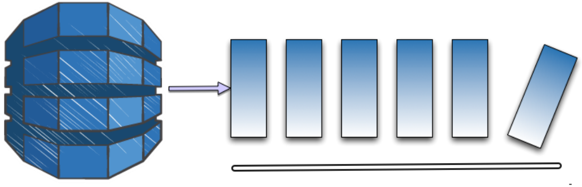
Status + Date

Take advantage of sparse indexes

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

DynamoDB Streams

DynamoDB Streams



Stream of updates to a table

Asynchronous

Exactly once

Strictly ordered

- Per item

Highly durable

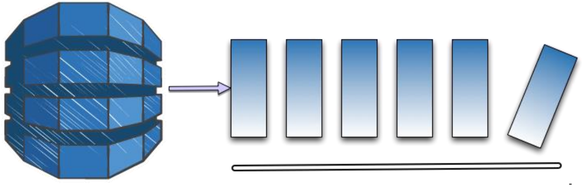
- Scale with table

24-hour lifetime

Sub-second latency

View types

UpdateItem (Name = John, Destination = Pluto)

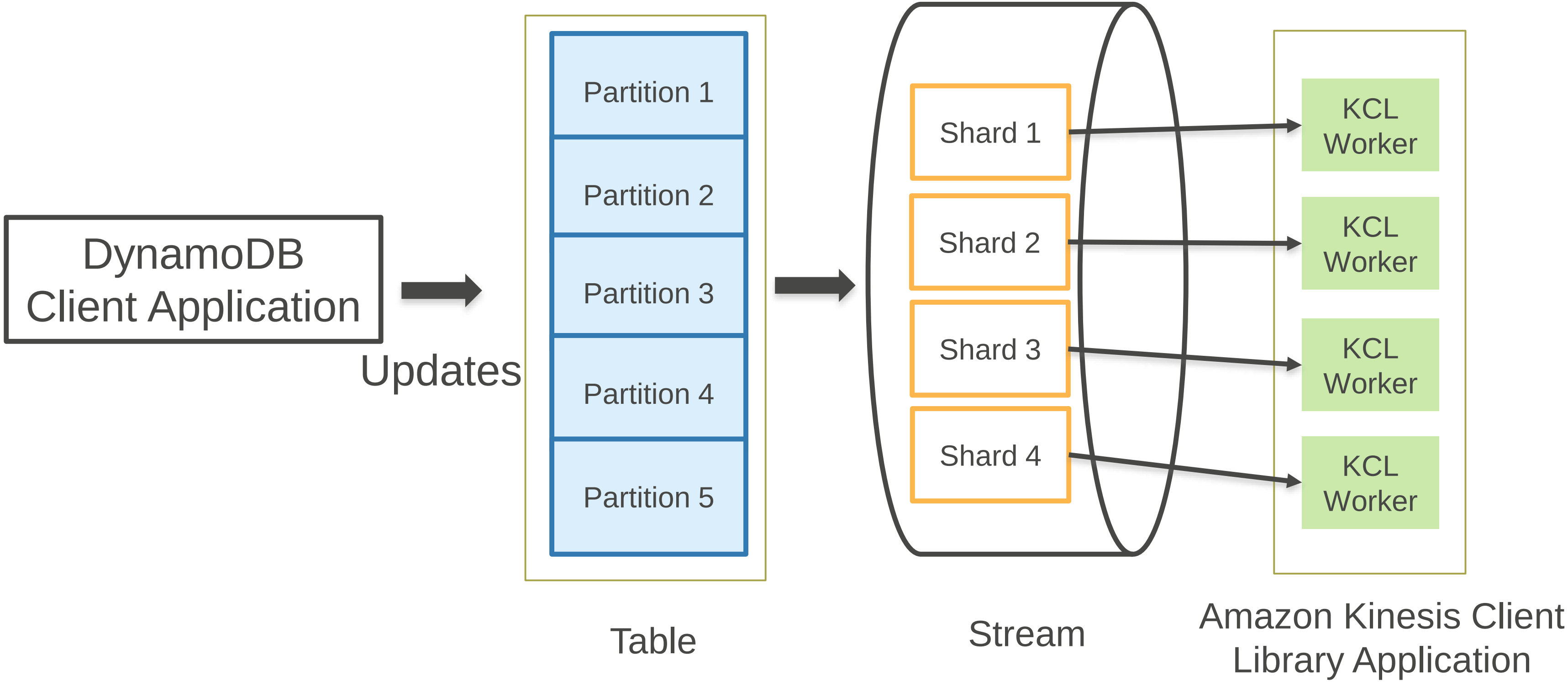


View Type	Destination
Old image—before update	Name = John, Destination = Mars
New image—after update	Name = John, Destination = Pluto

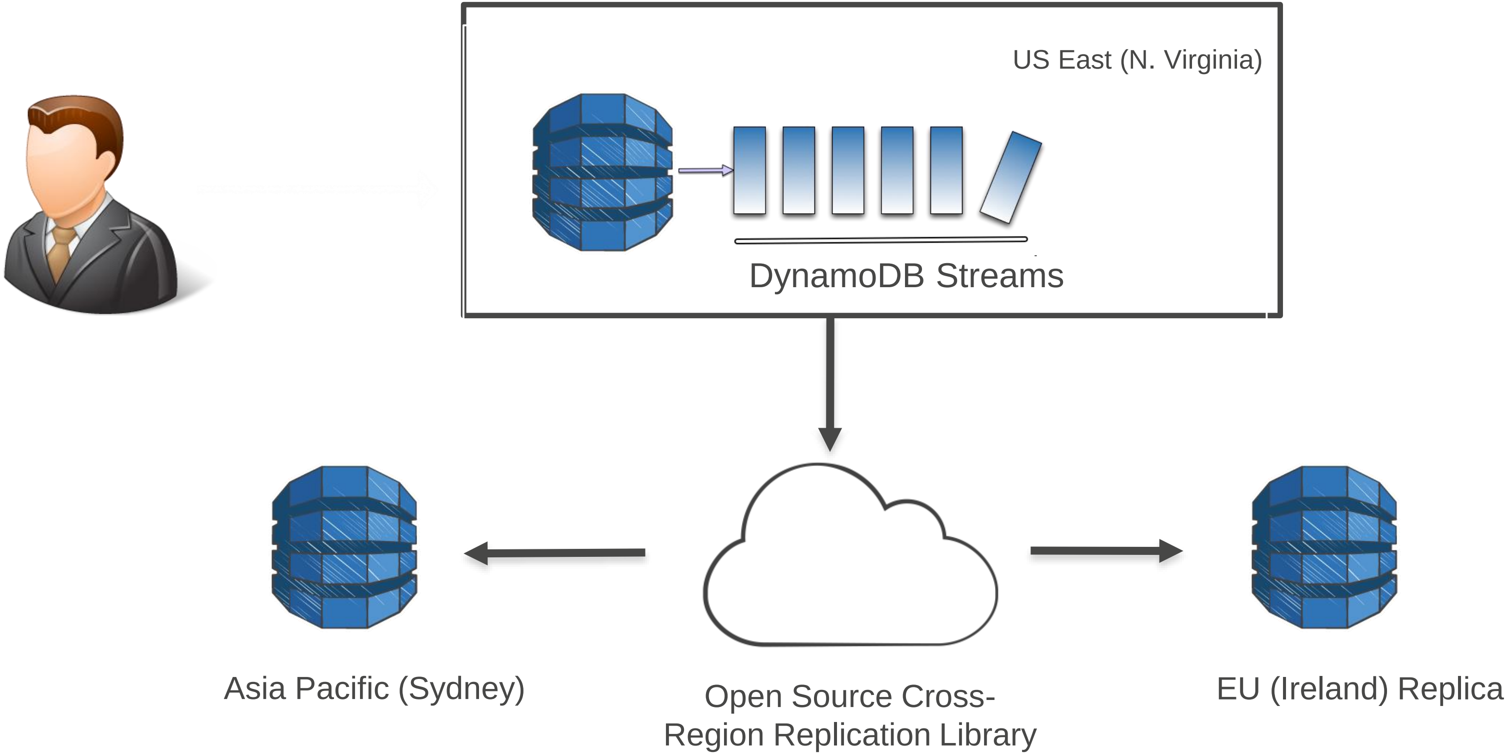
Old and new images	Name = John, Destination = Mars
	Name = John, Destination = Pluto

Keys only	Name = John

DynamoDB Streams and Amazon Kinesis Client Library



Cross-region replication



US East (N. Virginia)

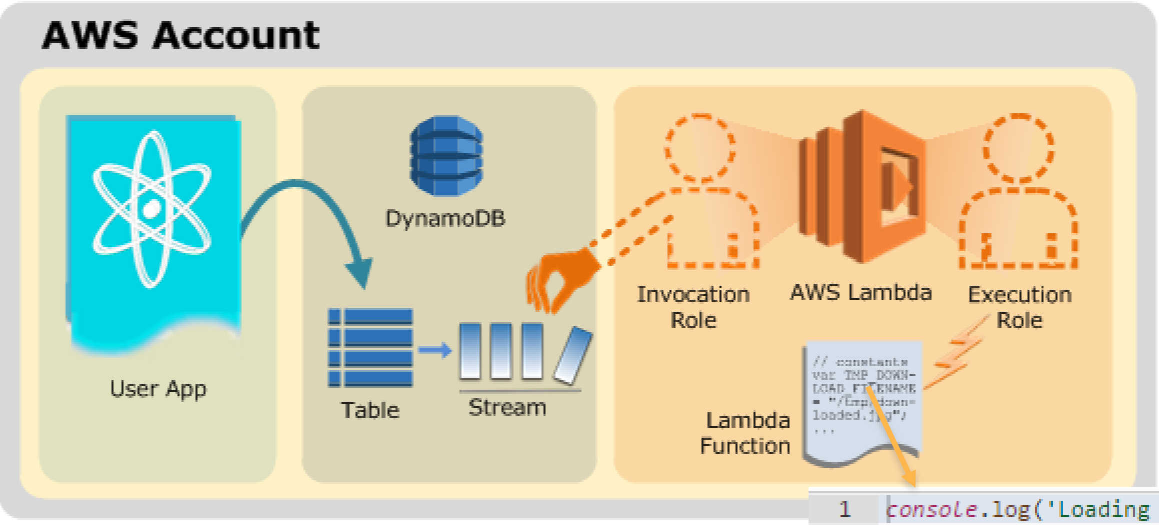
DynamoDB Streams

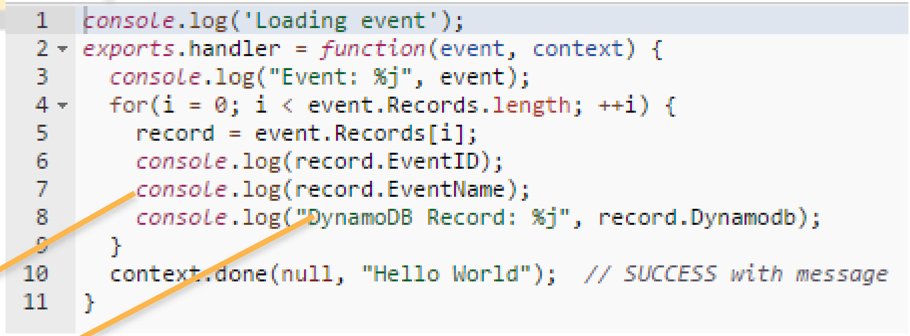
Asia Pacific (Sydney)

Open Source Cross-Region Replication Library

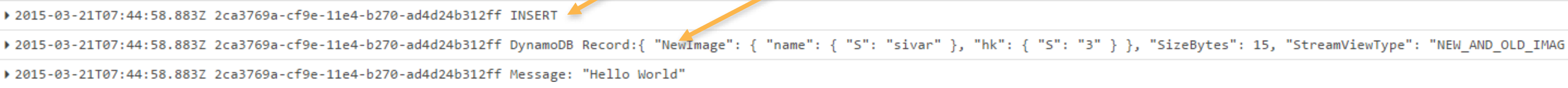
EU (Ireland) Replica

DynamoDB Streams and AWS Lambda



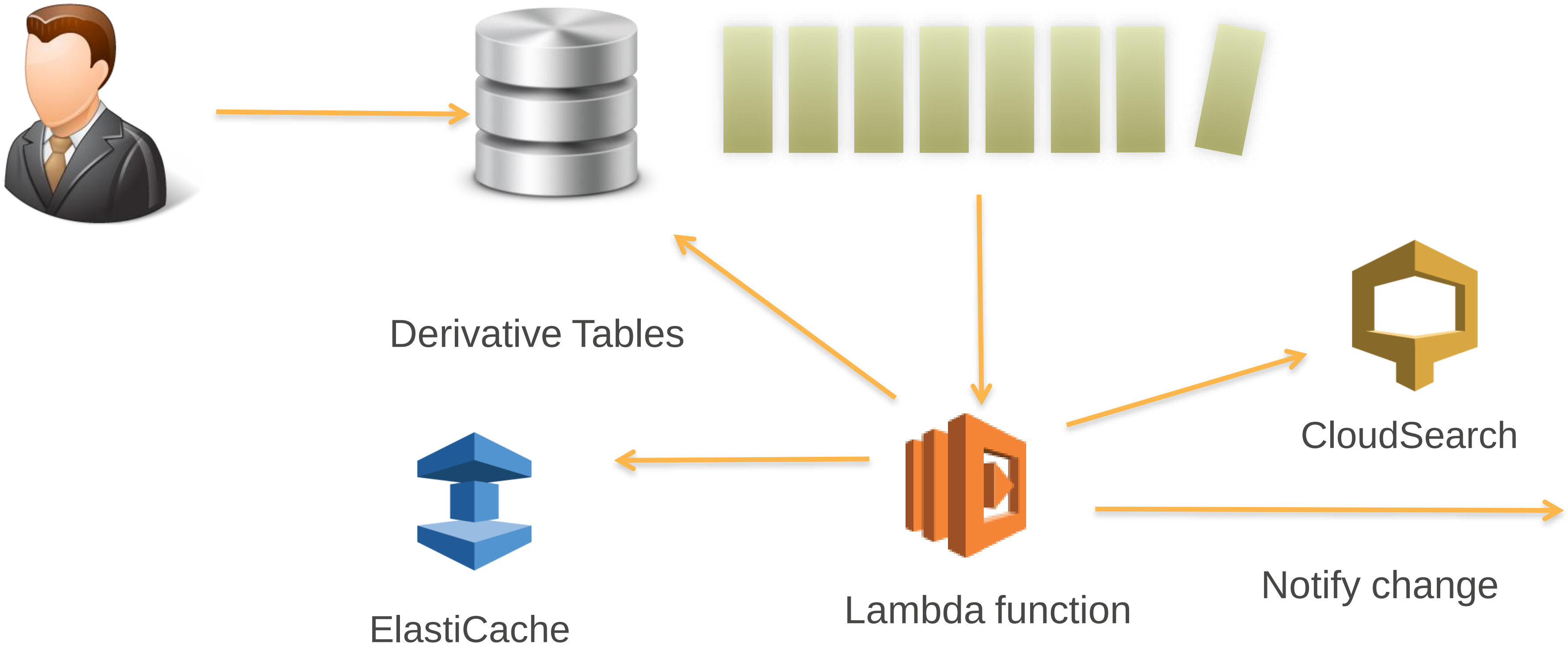


```
console.log('Loading event');
exports.handler = function(event, context) {
  console.log("Event: %j", event);
  for(i = 0; i < event.Records.length; ++i) {
    record = event.Records[i];
    console.log(record.EventID);
    console.log(record.EventName);
    console.log("DynamoDB Record: %j", record.Dynamodb);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



```
▸ 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"
```

Triggers



Derivative Tables

CloudSearch

Notify change

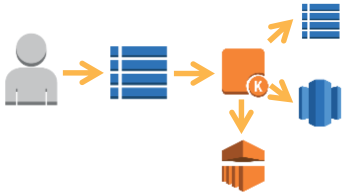
Lambda function

ElastiCache

☑ Analytics with DynamoDB Streams

Collect and de-dupe data in DynamoDB

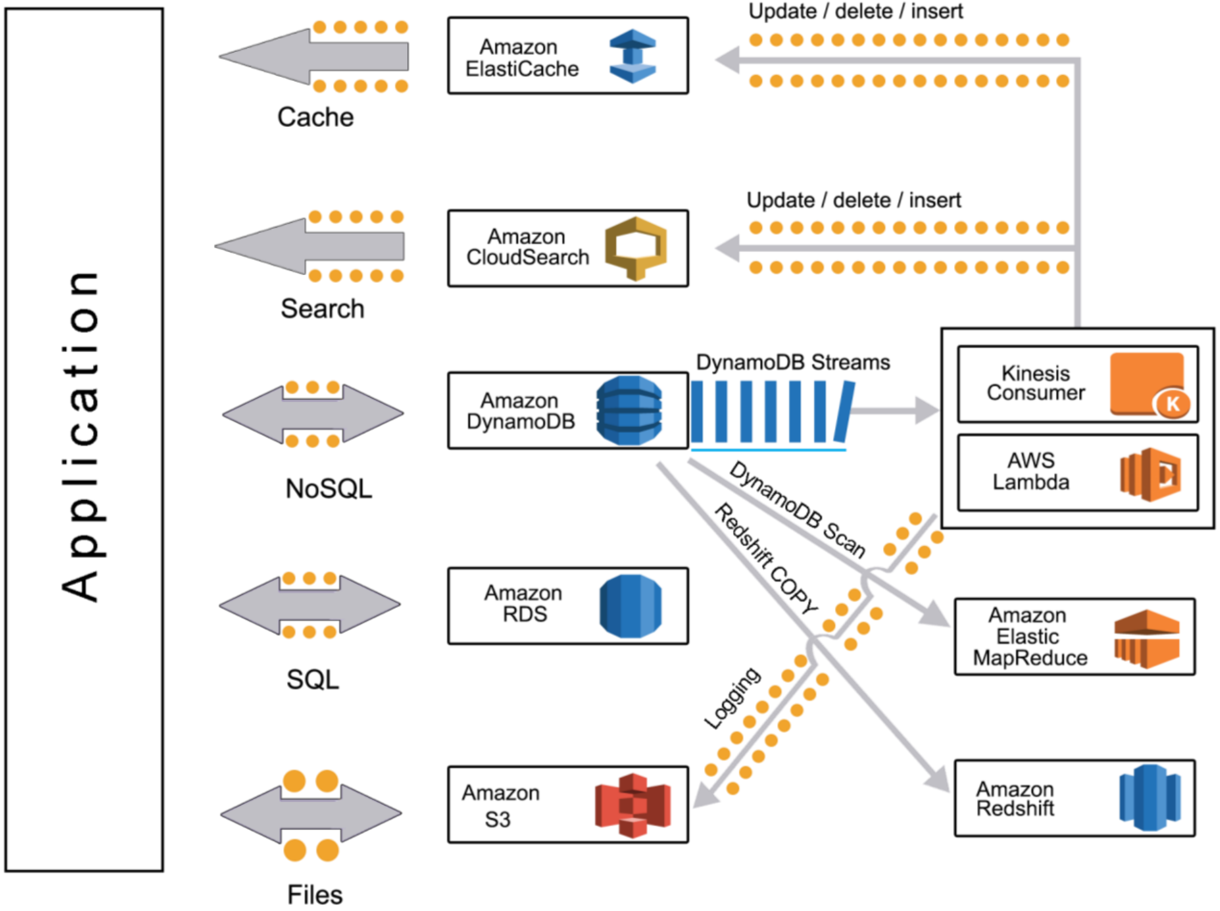
Aggregate data in-memory and flush periodically



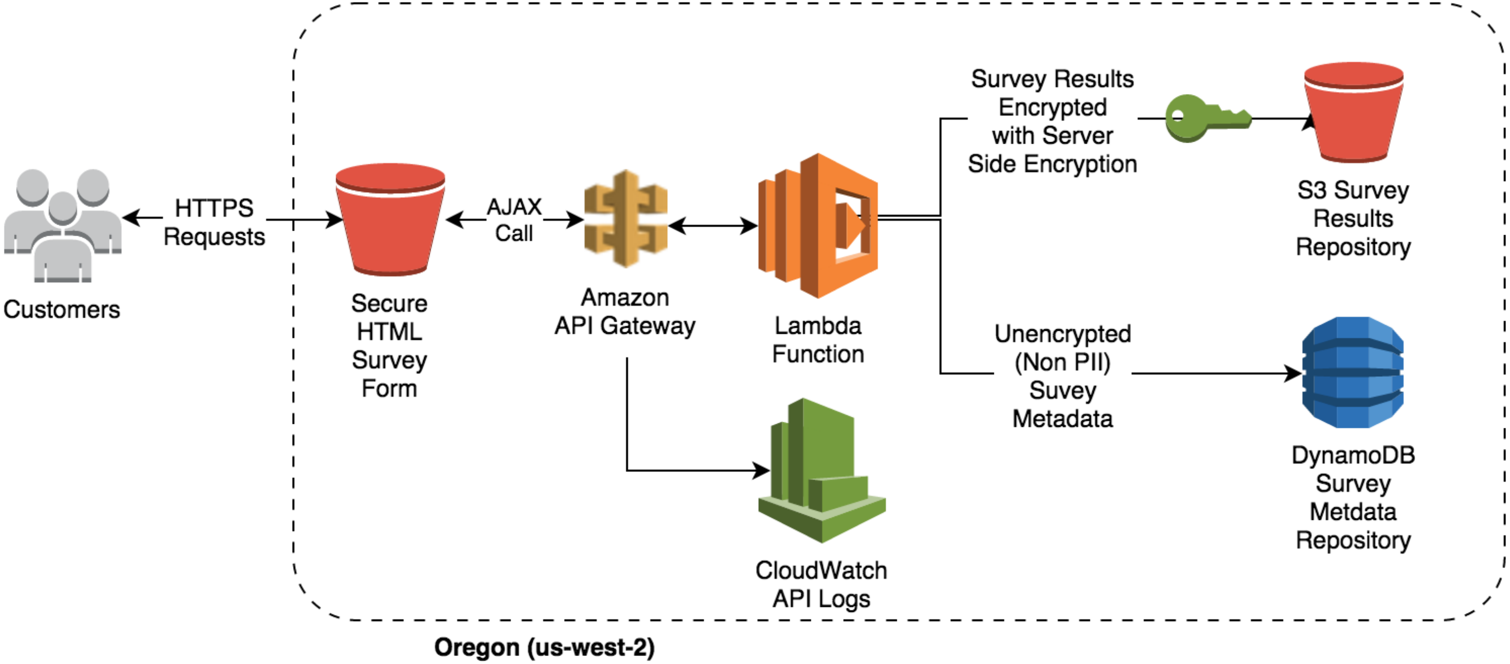
Important when: Performing real-time aggregation and analytics

Architecture

Reference Architecture



Elastic Event Driven Applications


Customers
HTTPS Requests
Secure HTML Survey Form
AJAX Call
Amazon API Gateway
Lambda Function
CloudWatch API Logs
Survey Results Encrypted with Server Side Encryption
S3 Survey Results Repository
Unencrypted (Non PII) Suvey Metadata
DynamoDB Survey Metdata Repository
Oregon (us-west-2)

Thank you!