Offline/real time use cases

Users expect data immediately

- Banking alerts
- News stories
- Multi-player games
- Chat applications
- Shared whiteboards
- AR/VR experiences
- Document collaboration

Users expect data availability offline

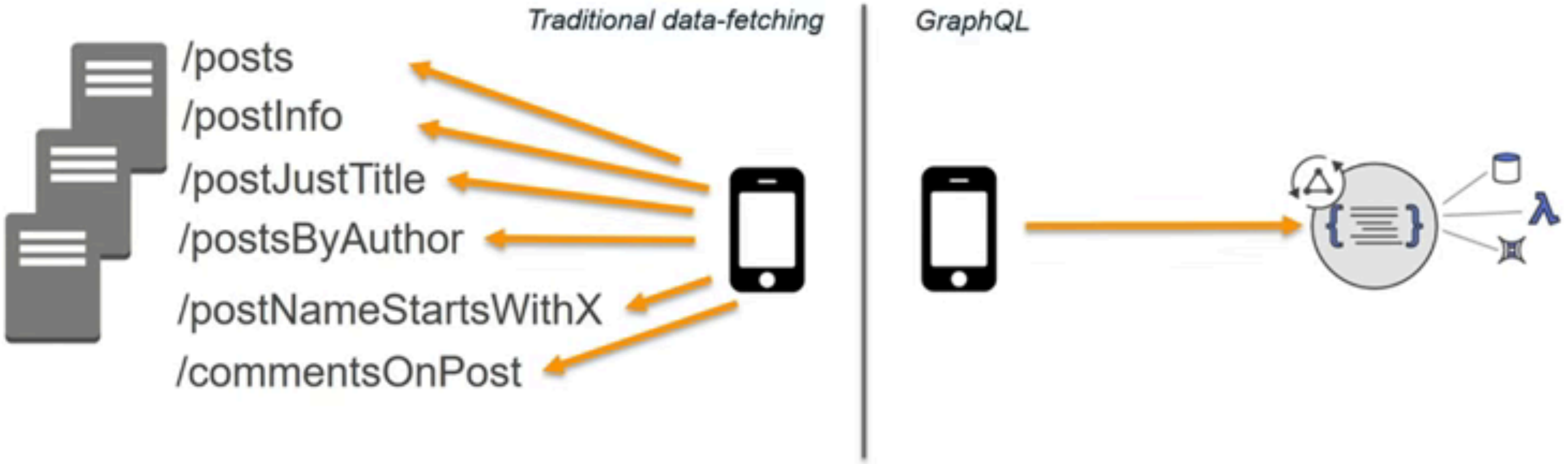
- Financial transactions
- News articles
- Games
- Messaging (pending chat)
- Document collaboration

What is GraphQL?

Open, declarative data-fetching specification

!= Graph database

Use NoSQL, Relational, HTTP, etc.



What are the GraphQL benefits?

Rapid prototyping and iteration

Introspection

Co-location of data requirements & application views

- Implementations aren’t encoded in the server

Data behavior control

- Batching, request/response and real-time

Bandwidth optimization (N+1 problem)

Can you do … with GraphQL?

Real time? YES

Batching? YES

Pagination? YES

Relations? YES

Aggregations? YES

Search? YES

Offline? YES

What is AWS AppSync?

Managed service for application data using GraphQL with real-time capabilities and an offline programming model

- Connect to resources in your account
- Make your data services in real time or offline
- Use AWS services with GraphQL
- Automatic sync, conflict resolution in the cloud
- Enterprise-level security features

Real time/offline with AWS AppSync

Integrates with the popular Apollo GraphQL client (https://github.com/apollographql)

- Multiple platforms and frameworks

Offline support

- Automatically persisted for Queries
- Write-through model for Mutations
- Optimistic UI

Conflict Resolution in the Cloud

- Optional client callback

GraphQL Subscriptions

- Event driven model
- Automatic WebSocket connection

Offline data rendering

https://aws.github.io/aws-amplify/

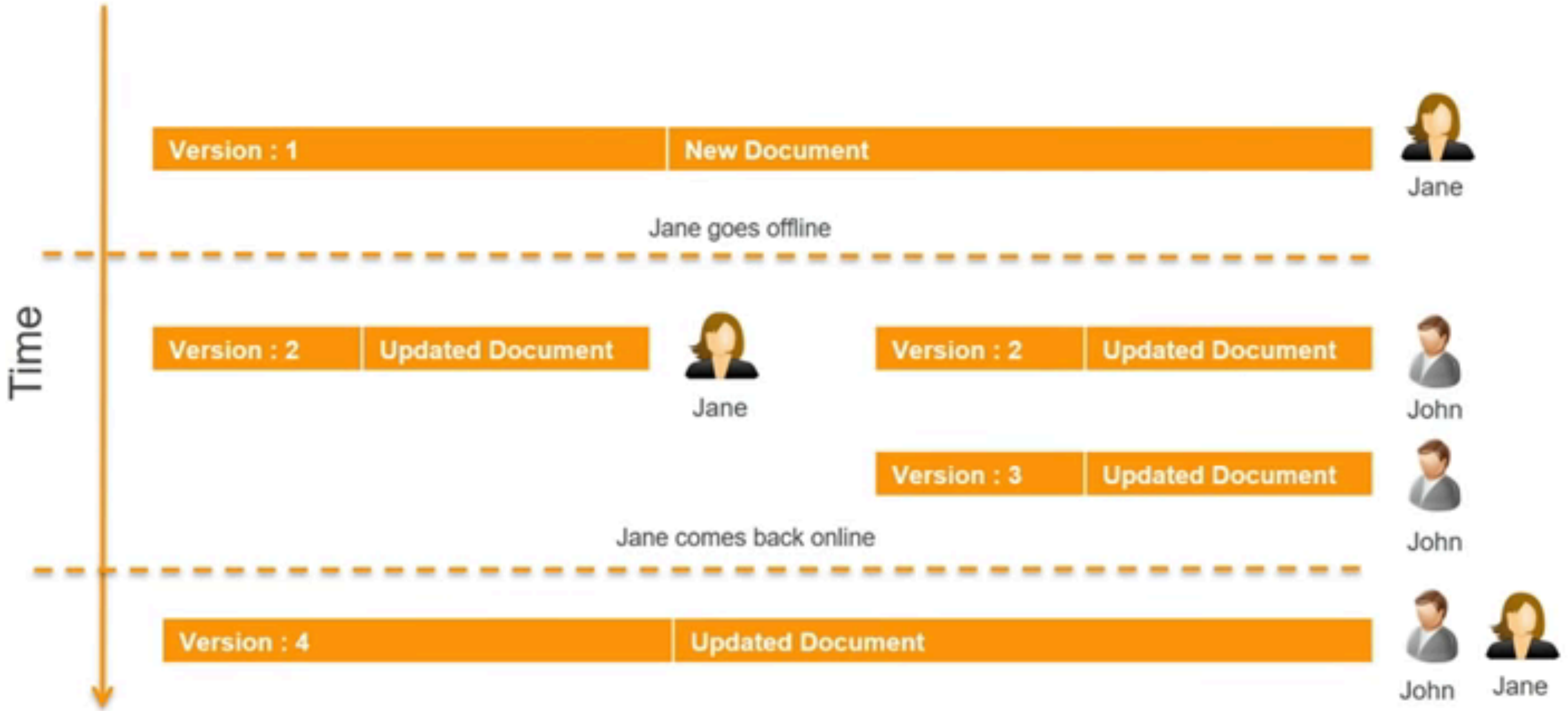
```
const client = new AWSAppSyncClient({
    url: awsconfig.ENDPOINT,
    region: AWS.config.region,
    auth: { type: AUTH_TYPE.AWS_IAM, credentials: Auth.currentCredentials() }
});

const WithProvider = () => (
    <ApolloProvider client={client}>
        <Rehydrated>
            <AppWithData />
        </Rehydrated>
    </ApolloProvider>
);
```

That's it! Data is automatically available offline!

AWS re:Invent

aws

Offline mutations



Optimistic UI

```
options: {
  fetchPolicy: 'cache-and-network'
},
props: (props) => ({
    onAdd: post => props.mutate({
        optimisticResponse: () => ({
           addPost: { __typename: 'Post', content: ‘New data!', version: 1, ...post }
        }),
     })
}),
update: (dataProxy, { data: { addPost } }) => {
    const data = dataProxy.readQuery({AllPostsQuery});
    data.posts.push(addPost);
    dataProxy.writeQuery({AllPostsQuery, data });
}}
```

AWS re:Invent

aws

Conflict Resolution and synchronization

Conflict resolution in the cloud

1. Server wins
2. Silent reject
3. Custom logic (AWS Lambda)

- Optimistic version check
- Extend with your own checks

Optional

- Client callback for Conflict Resolution is still available as a fallback

Example: Check that an ID doesn't already exist:

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   },
   "condition" : {
      "expression" : "attribute_not_exists(id)"
   }
}
```

Run Lambda if version wrong:

```
"condition" : {
    "expression" : "someExpression"
    "conditionalCheckFailedHandler" : {
        "strategy" : "Custom",
        "lambdaArn" : "arn:..."
    }
}
```

Client experience and configuration

Offline is a write-through "Store"

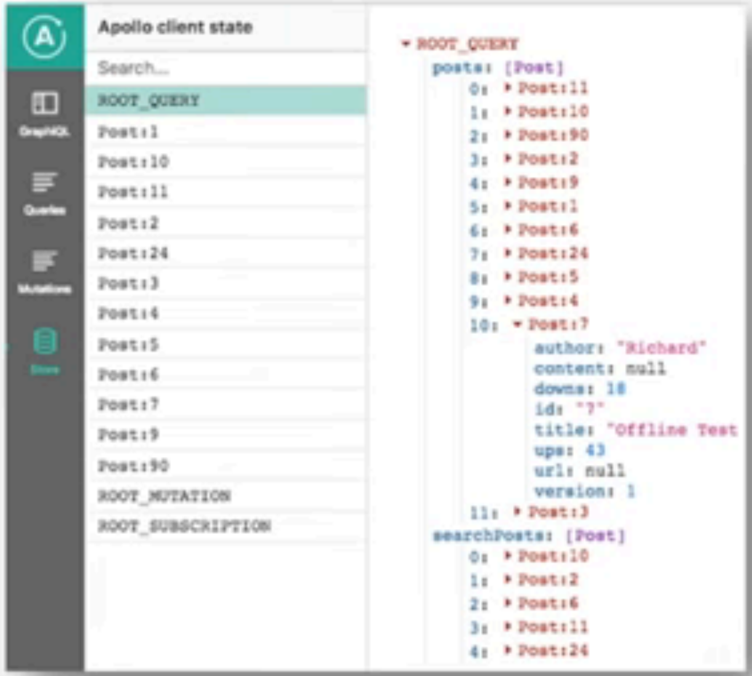
- Persistent storage mediums back the Apollo normalized cache
- Local Storage for web
- SQLite on hybrid/native platforms

SQLite database can be preloaded

- Hydrate after installing from AppStore

Offline client can be configured

- Wifi only vs. wifi & carrier



```
ROOT_QUERY
Post:1
Post:10
Post:11
Post:2
Post:24
Post:3
Post:4
Post:5
Post:6
Post:7
Post:9
Post:90
ROOT_MUTATION
ROOT_SUBSCRIPTION
```

```
▾ ROOT_QUERY
    posts: [Post]
        0: ▸ Post:11
        1: ▸ Post:10
        2: ▸ Post:90
        3: ▸ Post:2
        4: ▸ Post:9
        5: ▸ Post:1
        6: ▸ Post:6
        7: ▸ Post:24
        8: ▸ Post:5
        9: ▸ Post:4
        10: ▾ Post:7
              author: "Richard"
              content: null
              downs: 18
              id: "7"
              title: "Offline Test
              ups: 43
              url: null
              version: 1
        11: ▸ Post:3
    searchPosts: [Post]
        0: ▸ Post:10
        1: ▸ Post:2
        2: ▸ Post:6
        3: ▸ Post:11
        4: ▸ Post:24
```

Images and rich content

```
type S3Object {
  bucket: String!
  key: String!
  region: String!
}
```

```
type Profile {
  name: String!
  profilePic: S3Object!
}
```

```
input S3ObjectInput {
  bucket: String!
  key: String!
  region: String!
  localUri: String!
}
```

```
type Mutation {
  updatePhoto(name: String!,
              profilePicInput: S3ObjectInput!): Profile
}
```

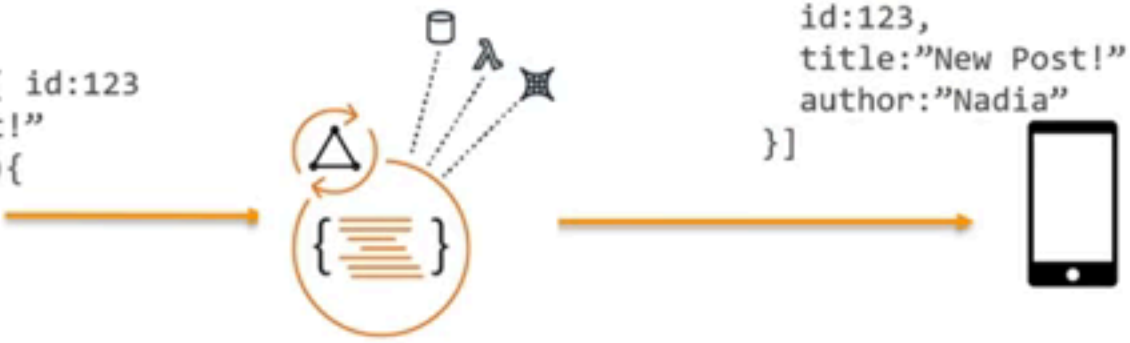
GraphQL Subscriptions

Near real time updates of data

Event based mode, triggered by Mutations
- Scalable model, designed as a platform for common use-cases

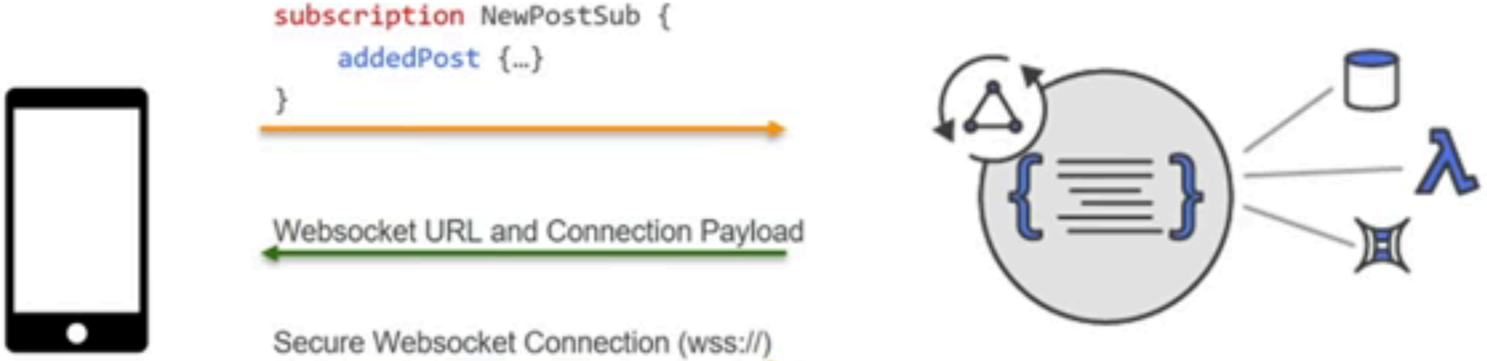
Can be used with ***ANY*** data source in AppSync
- Lambda, DynamoDB, Elasticsearch

```
mutation addPost( id:123
  title:"New post!"
  author:"Nadia"){
  id
  title
  author
}
```



```
data: [{
  id:123,
  title:"New Post!"
  author:"Nadia"
}]
```

Handshake process



```
subscription NewPostSub {
    addedPost {…}
}
```

Websocket URL and Connection Payload
Secure Websocket Connection (wss://)

Schema directives

```
type Subscription {
    addedPost: Post
    @aws_subscribe(mutations: ["addPost"])
    deletedPost: Post
    @aws_subscribe(mutations: ["deletePost"])
}


type Mutation {
    addPost(id: ID! author: String! title:
     String content: String): Post!
    deletePost(id: ID!): Post!
}
```

```
subscription NewPostSub {
    addedPost {
        __typename
        version
        title
        content
        author
        url
    }
}
```

Real time UI updates

```
const AllPostsWithData = compose(
  graphql(AllPostsQuery, { options: { fetchPolicy: 'cache-and-network' },
    props: (props) => ({
      posts: props.data.posts,
      subscribeToNewPosts: params => {
          props.data.subscribeToMore({
              document: NewPostsSubscription,
              updateQuery: (prev, { subscriptionData: { newPost } }) => ({
                ...prev,
                posts: [newPost, ...prev.posts.filter(post => post.id !== newPost.id)]
              })
            });
        });
    …//more code
```

Best practices

Don't boil the ocean – start with offline for Queries

Mutations offline – what UIs actually need to be optimistic?

Use Subscriptions appropriately
- Large payloads/paginated data: Queries
- Frequent updating deltas: Subscriptions
- Be kind to your customer's battery & CPU!

Don't overcomplicate Conflict Resolution
- Data model appropriately, many app actions simply append to a list
- For custom cases, use a AWS Lambda and keep client logic light (race conditions)