Building a Scalable Event Service with Cassandra: Design to Code

Tareq Abedrabbo - OpenCredo
Code Mesh 2014
About Me

• CTO at OpenCredo

• We are a software consultancy and delivery company

• Open source, NoSQL/Big Data, cloud
This talk is about…

- What we built
- Why we built it
- How we built it
Project background
• High street retailer
• Decoupled micro services architecture
• Java-based, event-driven platform
• Cassandra, Cloud Foundry, RabbitMQ
Why do we need an event service?
• Capture millions of platform and business events
• Trigger downstream processes asynchronously
• Customise standard processes in a non-intrusive way
• Provide a system-wide transaction log
• Analytics
• System testing
I know, I will use technology X!
However...

- Ambiguous requirements
- New paradigm, emerging architecture
- We need to look at the problem as a whole
- We need to avoid building useless features
- We need to avoid accumulating technical debt
Design principles
1. Simplicity (yes, really!)
Edward de Bono

Simplicity
2. Decoupling
• Contract-first design

• Flexibility in the implementation

• Ability to evolve while minimising impact of changes
3. Scalability and fault-tolerance
• Choosing the right architecture
• Choosing the right model
• Choosing the right tools
What is an event?
• A **simple** event is an opaque value, typically a time series item
  
  • meter reading

• A **structured** event can have an arbitrarily complex structure that can evolve over time
  
  • user registration event
What does the event store look like?
Event service API, version 1: store and read an event
• It needs to be **simple** and **accessible**

• a service only cares about emitting events

• at that stage, we didn’t care much about the structure of each individual event

• accessible ideally even from outside the platform

• Resource oriented design - **ReST**

• Simple request/response paradigm
• Store an event
  • POST /api/events/

• Read an event
  • GET /api/events/{eventId}
Anatomy of an Event

{
    "type" : "DEMOENTITY.DEMOPROCESS.DEMOTASK",
    "source" : "demoapp1:2.5:981a24b3-2860-40ba-90d4-c47ef1a70abe",
    "clientTimestamp" : 1401895567594,
    "serverTimestamp" : 1401895568616,
    "platformContext" : {
        "id" : "demoapp1",
        "version" : "2.5"
    },
    "businessContext" : {
        "channel" : "WEB",
    },
    "payload" : {
        "message" : "foo",
        "anInteger" : 33,
        "bool" : false
    }
}
and the architecture to support the requirements...
The Event Table

<table>
<thead>
<tr>
<th>Key</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>timestamp</td>
</tr>
<tr>
<td>id1</td>
<td>123</td>
</tr>
<tr>
<td>id2</td>
<td>456</td>
</tr>
</tbody>
</table>
• Store payload as a blob
• Established service minimal contract
• Established semantics: POST
Event service API, version 2: querying events and notifications
• Query events
  • GET /api/events?{queryString}
  • {queryString} can consist of the following fields:
    • start, end, startOffset, limit, tag, type, order
• Examples:

  • GET /api/events?start={startTime} &end={endTime}

  • GET /api/events?
    startOffset=3600000&type=someType
How do we model time series?
Simple Time Series Modelling

Using timestamps as a clustering column

<table>
<thead>
<tr>
<th>Key</th>
<th>Timestamp/Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>ts11, ts12, ts13</td>
</tr>
<tr>
<td></td>
<td>v11, v12, v13</td>
</tr>
<tr>
<td>id2</td>
<td>ts21, ts22, ts23</td>
</tr>
<tr>
<td></td>
<td>v21, v22, v23</td>
</tr>
</tbody>
</table>
• **Pros**
  
  • Simple
  
  • Works well for simple data structures
  
  • Good read and write performance

• **Cons**
  
  • Hard limit on partition size (2 billion cells)
  
  • Limited flexibility
  
  • Limited querying
## Time Bucketing

Adding a time bucket to the partition key

<table>
<thead>
<tr>
<th>Key</th>
<th>Timestamp/Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>ts11, ts12, ts13</td>
</tr>
<tr>
<td>id1</td>
<td>v11</td>
</tr>
<tr>
<td>id1</td>
<td>bucket1, bucket2</td>
</tr>
<tr>
<td>id2</td>
<td>ts11, ts12, ts13</td>
</tr>
<tr>
<td>id2</td>
<td>bucket1, bucket2</td>
</tr>
<tr>
<td></td>
<td>v12, v13</td>
</tr>
<tr>
<td></td>
<td>v21, v22</td>
</tr>
<tr>
<td></td>
<td>v23</td>
</tr>
</tbody>
</table>
• Mitigates the partition size issue
• Queries become slightly more complex
• Write performance is not affected
• Reads may be slower, potentially hitting multiple buckets
How about querying?
### Querying

One denormalised table for each query

<table>
<thead>
<tr>
<th>Query Key</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id1</td>
<td>ts11</td>
<td>id1</td>
<td>ts12</td>
<td>id2</td>
<td>ts21</td>
<td>id2</td>
</tr>
<tr>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **p1**: b1 | v11
- **p2**: b2 | v12 | v21
- **p2**: b2 | v22
• Denormalise for each query

• Higher disk usage

• Disk space is cheap, but not free

• Write latency is affected

• Time-bucketed indexes can create hot spots (hot shards)
There is obviously no optimal solution...
Event Store

Event Service
• Same service contract

• Basic client guarantee: if a POST is successful the event has been persisted “sufficiently”

• Indices are updated asynchronously

• Events can be published to a message broker
This is actually CQQRS
Evolution of Event

- Payload and meta-data as simple collections of key/value
- The type is persisted with each event
  - to make events readable
  - to avoid managing schemas
Primary Event Store

Events are simply keyed by id

events (id timeuuid primary key,
    source text, type text, cts timestamp,
    sts timestamp, bct map<text, text>,
    bcv map<text, blob>,
    pct map<text, text>,
    pcv map<text, blob>,
    plt map<text, text>,
    plv map<text, blob>);

Indices

• Ascending and descending time buckets for each query type

• Index value ‘points’ to an event stored in the main table
Indices

Ascending and descending time buckets for each query type

events_by_time_asc (  
tbucket text, eventid timeuuid,  
primary key (tbucket, eventid))  
with clustering order by (eventid asc);

events_by_time_desc (  
tbucket text, eventid timeuuid,  
primary key (tbucket, eventid))  
with clustering order by (eventid desc);
Implementing Pagination
# Pagination

```plaintext
GET /api/events?start=141..&type=X&limit=5
```

<table>
<thead>
<tr>
<th>type</th>
<th>time</th>
<th>id1</th>
<th>id2</th>
<th>id3</th>
<th>id4</th>
<th>id5</th>
<th>id6</th>
<th>id7</th>
<th>id8</th>
</tr>
</thead>
<tbody>
<tr>
<td>type1</td>
<td>bucket1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type1</td>
<td>bucket2</td>
<td>id1</td>
<td>id2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type1</td>
<td>bucket3</td>
<td></td>
<td></td>
<td>id3</td>
<td>id4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type1</td>
<td>bucket4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>id5</td>
<td>id6</td>
<td>id7</td>
<td>id8</td>
</tr>
<tr>
<td>type1</td>
<td>bucket5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Pagination

**GET** /api/events?start=141..&type=X&limit=5

<table>
<thead>
<tr>
<th>type</th>
<th>time ▶</th>
<th>type1 bucket1</th>
<th>type1 bucket2</th>
<th>id1</th>
<th>id2</th>
<th>type1 bucket3</th>
<th>id3</th>
<th>id4</th>
<th>type1 bucket4</th>
<th>id5</th>
<th>id6</th>
<th>id7</th>
<th>id8</th>
<th>type1 bucket5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>query</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>range</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


### Pagination

**GET** /api/events?start=141..&type=X&limit=5

<table>
<thead>
<tr>
<th>type</th>
<th>time</th>
<th>query range</th>
</tr>
</thead>
<tbody>
<tr>
<td>type1</td>
<td>bucket1</td>
<td></td>
</tr>
<tr>
<td>▲</td>
<td>type1</td>
<td>bucket2</td>
</tr>
<tr>
<td>type1</td>
<td>bucket3</td>
<td></td>
</tr>
<tr>
<td>type1</td>
<td>bucket4</td>
<td></td>
</tr>
<tr>
<td>type2</td>
<td>bucket5</td>
<td></td>
</tr>
</tbody>
</table>
## Pagination

**GET /api/events?start=141..&type=X&limit=5**

<table>
<thead>
<tr>
<th>type</th>
<th>time</th>
<th>query range</th>
</tr>
</thead>
<tbody>
<tr>
<td>type1</td>
<td>bucket1</td>
<td></td>
</tr>
<tr>
<td>type1</td>
<td>bucket2</td>
<td>id1</td>
</tr>
<tr>
<td>type1</td>
<td>bucket3</td>
<td>id3</td>
</tr>
<tr>
<td>type1</td>
<td>bucket4</td>
<td>id5</td>
</tr>
<tr>
<td>type2</td>
<td>bucket5</td>
<td></td>
</tr>
</tbody>
</table>
### Pagination

```
GET /api/events?start=141..&type=X&limit=5
```

<table>
<thead>
<tr>
<th>type</th>
<th>time</th>
<th>query range</th>
</tr>
</thead>
<tbody>
<tr>
<td>type1</td>
<td>bucket1</td>
<td></td>
</tr>
<tr>
<td>type1</td>
<td>bucket2</td>
<td>id1, id2</td>
</tr>
<tr>
<td>type1</td>
<td>bucket3</td>
<td>id3, id4</td>
</tr>
<tr>
<td>type1</td>
<td>bucket4</td>
<td>id5, id6, id7, id8</td>
</tr>
<tr>
<td>type2</td>
<td>bucket5</td>
<td></td>
</tr>
</tbody>
</table>

**Query Range:**

- **Start:** 141
- **Limit:** 5
```json
{
    "count": 1,
    "continuation": "http://event-service-location/api/events?continueFrom=9f00e9d0-ebfc-11e3-81c5-09597ebbf2cb&end=1401965827000&limit=1",
    "events": [
        {
            "type": "DEMOENTITY.DEMOPROCESS.DEMOTASK",
            "source": "demoapp1:2.5:981a24b3-2860-40ba-90d4-c47ef1a70abe",
            "clientTimestamp": 1401895567594,
            "serverTimestamp": 1401895568616,
            "platformContext": {
                "id": "demoapp1",
                "version": "2.5"
            },
            "businessContext": {
            },
            "payload": {
            }
        }
    ]
}
```
Pro

• Decoupling
  • client are unaware of the implementation details

• Intuitive ReSTful interface

• Disk consumption is more reasonable

• Easily extensible
  • Pub/sub
• Cons
  • Not only optimised for latency
    • Still sufficiently performant for our use-cases
  • More complex service code
  • Needs to execute multiple CQL queries in sequence
  • Cluster hotspots can still occur, in theory
Where do we go from here?
• Data model improvements: User Defined Types
• More sophisticated error handling
• Analytics with Spark
• Add other data views
Lessons learnt

• Scalability is not only about raw performance
• Experiment
• Simplify
• Understand Thrift, use CQL
Links

• OpenCredo: http://www.opencredo.com/blog

• Twitter: @tareq_abedrabbo

Thank you! Questions?