

Couchbase Eventing

Now officially named! - Couchbase Functions

[Introduction](#)

[Features](#)

[Design](#)

[Manager module \(MM\)](#)

[JavaScript execution engine\(JSEE\)](#)

[Communication between MM and JSEE](#)

[Eventing Process Architecture](#)

[Super supervisor](#)

[Producer](#)

[Consumer](#)

[Workflow](#)

[Rebalance](#)

[Topology awareness](#)

[Eventing Rebalance](#)

[Prepare Phase](#)

[Start Phase](#)

[Operations prohibited during eventing rebalance](#)

[Writing event handlers](#)

[Timer Events](#)

[Assurance](#)

[Failure cases of timers](#)

[Failure Handling of timers](#)

[TMB rollback](#)

[CMB rollback](#)

[TMB blob is lost](#)

[CMB is lost:](#)

[Wall clock time is skewed:](#)

[KV auto-failover is turned off](#)

[Special considerations for Timer Events](#)

[ns_server integration](#)

[Transpilation](#)

[Transpilation features](#)

[Using N1QL as part of JavaScript](#)

[Variable substitution](#)

[Iterator](#)

[N1QL Query API](#)

[Constructor](#)

[Properties](#)

[Instance methods](#)

[Transpilation components](#)

[Pre - processing](#)

[Phase - 1](#)

[Phase - 2](#)

[Transpilation Stage](#)

[Execution Stage](#)

[Iterator mechanism](#)

[Entry criteria](#)

[Exit criteria](#)

[Unlabelled break](#)

[Unlabelled continue](#)

[Labelled break](#)

[Labelled continue](#)

[Return](#)

[Throw](#)

[Uncaught exceptions](#)

[Nested iterators](#)

[Bubbling](#)

[Bubbling design](#)

[Optimizations](#)

[Optional query execution](#)

[Streaming cancellation](#)

[Constraints](#)

[Disallow conflicting identifiers](#)

[Disallow anonymous functions](#)

[N1QL statement declaration](#)

[Variable substitution in N1QL statements](#)

[Failure Behavior](#)

[Hard failover of eventing node\(s\)](#)

[Hard failover of KV node\(s\) with KV replicas](#)

[Differentiating CE vs EE](#)

[Dependencies](#)

[Internal dependencies:](#)

[Third Party Dependencies:](#)

[Open questions:](#)

[Framework](#)

[Editor/UI](#)

[Upgrade](#)

Introduction

Eventing is intended to allow associating user code with any event that occurs inside Couchbase Server. An event is a change in state of any element of the server.

Eventing is a MDS enabled service, and will run user supplied code on nodes designated with Eventing role. The role supports linear scalability and online rebalance. It uses Google V8 to run user supplied JavaScript code. It doesn't support full JavaScript syntax, because the programming model we offer needs to automatically parallelize on multiple nodes to handle the volume of events. For example, no global variables are accessible in event handlers.

We add a number of extensions to JavaScript to make it easy to work with Couchbase. For example, Couchbase Buckets appear as JavaScript maps, N1QL results can be iterated over using the JavaScript 'for ... of ...' construct and a number of added functions allow event handlers to send messages, raise more events etc.

Features

- Notifications for events happening within key-value store within Couchbase i.e. DCP_MUTATION, DCP_DELETE and DCP_EXPIRY.
- Support for timed events i.e. trigger user supplied code at specific time in future (we intend to support at-least once model for firing of timer events)
- Support for interacting with external system (not Couchbase) via curl and a pub/sub queuing endpoint (under review with PM team).
- Horizontal scalability based on MDS in order to keep up with incoming events from key-value store within couchbase.

Design

Eventing instance running on any node with eventing node, can be divided into two sub-components broadly:

- Manager module
- JavaScript execution engine.

Manager module (MM)

- Maintain multiple different [event handlers](#) that user has deployed on the cluster.
- Allow Eventing nodes to scale up horizontally using MDS.
- Distribution of equal work across all eventing nodes in the cluster.
- Support rebalance/failover of nodes with Eventing role.
- Expose supervision tree model to manage different processes and routines.

Event handler:

Event handler is the code that a user has supplied that runs when a specific event occurs. On an eventing cluster, user could deploy multiple event handlers - each listening to events happening within a specific source bucket. User could also create multiple event handlers, each listening on same source bucket but each handler running different code on events.

JavaScript execution engine(JSEE)

- Execute user supplied JavaScript code for different events.
- Provides a mechanism to process the syntactic sugar for JavaScript for writing the event handler code.

We've chosen Golang, for implementing [MM](#), because it allows us to reuse modules written by other Couchbase components, and C++ for JSEE, because V8 is written in C++.

Communication between MM and JSEE

Communication between MM and JSEE happens over localhost TCP port, instead of CGO communication. Reasons for making that choice:

- Debugging CGO crashes is difficult.
- Given JSEE is running user supplied code, bugs in user code leading to V8 Engine crash shouldn't crash Golang runtime as well.
- As mentioned in [event handler](#), multiple event handlers can run on an eventing node. With CGO approach - crash in one event handler would all crash other event handlers.

To handle above concerns, JSEE associated with each event handler is a separate OS process - which interacts with MM via localhost TCP port.

Eventing Process Architecture

Below is architecture diagram of different eventing sub-components on one single eventing node (running two different applications).

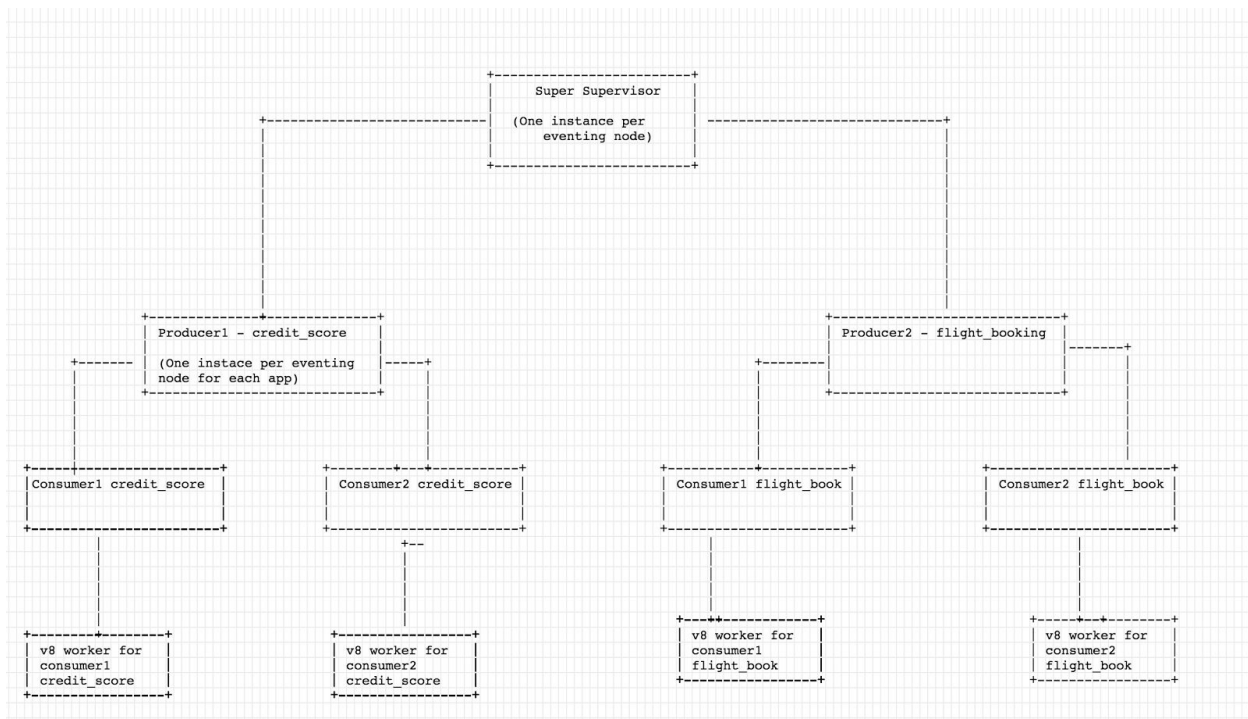


Fig 1: Eventing process architecture

Super supervisor

- There is a **super_supervisor** routine that runs one per eventing node. Babysitter process from ns_server is responsible for spawning this routine. If the super_supervisor crashes, babysitter process is responsible for respawning it.
- Super_supervisor's job is to manage all event handlers that are deployed on a given eventing node. It also monitors each **event handler management routine**(called "**Producer**") i.e. in-case any of the handler management routine crashes, super_supervisor is respawning it.

Producer

- There is just one running instance of **event handler management routine**(called "**Producer**" from here on) for each deployed event handler on a given node with eventing role.
- Producer routine is responsible for creating planner, which basically allow division of equal work across all eventing nodes.
- Producer routine spawns multiple worker routines, which are responsible for passing along different events to JavaScript world for triggering various handler code deployed by user.
- Producer also supervises each worker routine(called "**Consumer**") i.e. if any Consumer routine crashes or dies - Producer will respawn it.

Consumer

- Each consumer routine is tied up with V8 worker process in 1:1 fashion. They talk to each other over localhost TCP port. Consumer routine is responsible for supervising V8 worker process as well, if it crashes or dies - Consumer routine is responsible for respawning it.
- Each consumer spawns one DCP connection to each KV service node in the cluster.

The number of Consumer routine instances and V8 worker process can be changed dynamically when the end user wants to scale up event processing throughput on a node with eventing role.

Workflow

- End-users who want to leverage features provided by Eventing service would need to add one or more nodes with Eventing MDS role into the existing cluster. Then the babysitter process from cluster manager will spawn up *eventing* process on those node(s).
- Once the newly added eventing service node(s) are up - then under the “Eventing” tab in UI, end user could create new “Event Handlers”. Below is screen capture of UI, where *credit_score* Event Handler has been deployed.

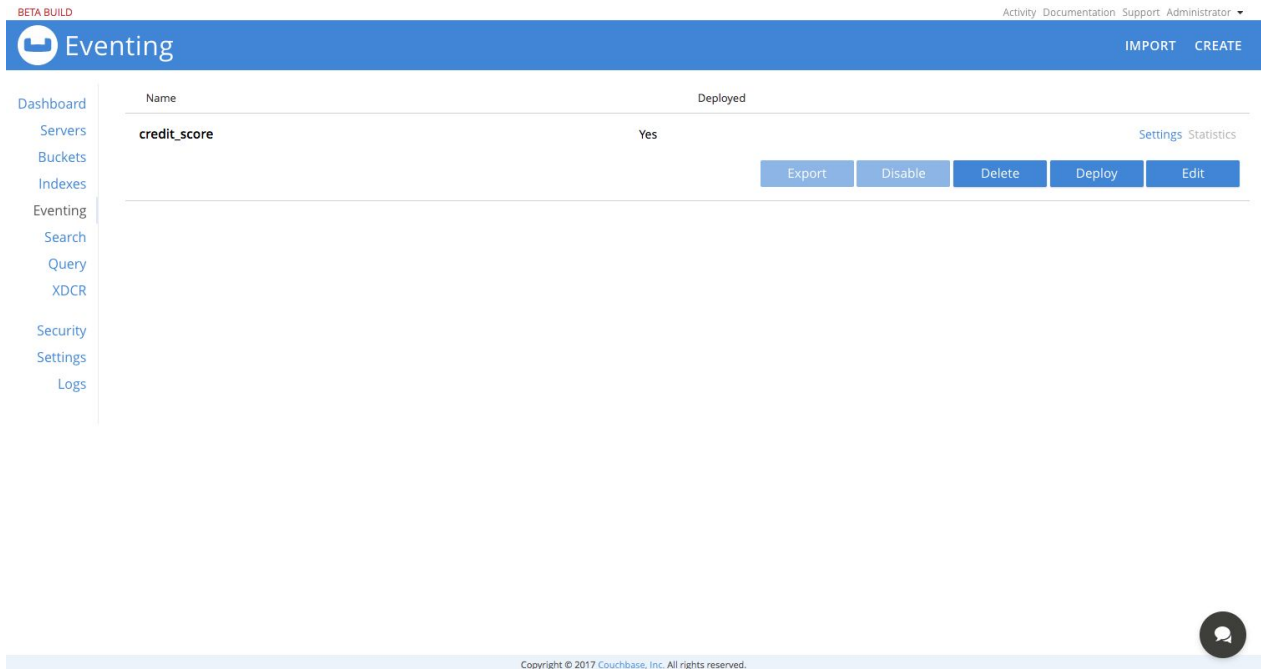


Fig 2: Eventing Handlers landing page

- Every event handler has two different components:
 - **Handlers** - Contains the Event Handler code(Fig 3)



Dashboard

Servers

Buckets

Indexes

Eventing

Search

Query

XDCR

Security

Settings

Logs

```

1- function OnUpdate(doc, meta) {
2   log("doc: ", doc, " meta: ", meta);
3
4-   switch(doc.type) {
5     case "credit_score":
6       updated_doc = CalculateCreditScore(doc);
7       credit_bucket[meta.docid] = updated_doc;
8
9       var value = credit_bucket[meta.docid];
10      //delete credit_bucket[meta.docid];
11      break;
12
13     case "travel_sample":
14       log("Got travel sample blob", doc);
15       break;
16
17     case "cpu_op":
18       var i;
19       for (i = 0; i < 1000 * 1000;) {
20         i++;
21       }
22       log("Final count:", i);
23       break;
24
25     case "doc_timer":
26       docTimer(timerCallback, meta.docid, meta.expiry);
27       break;
28
29     case "non_doc_timer":
30       nonDocTimer(NDtimerCallback, meta.expiry);
31       break;
32
33     case "query":
34       var bucket = `beer-sample`;
35       var limit = 5;
36       var type = "brewery";
37
38       var n1qlResult = SELECT name
39                       FROM :bucket
40                       WHERE type == ':type'

```

Debug ⓘ

Cancel

Save

☒ Deploy & Save

Fig 3: Event Handler Code

- **Deployment Plan** - Captures Event Handler related settings(Fig 4):
 - source_bucket - Bucket from where Event Handler is interested in listening to events from. Eventing nodes will open up DCP streams from source_bucket in order to listen on events happening within it.
 - metadata_bucket - Bucket where the “Eventing service” stores metadata information about timer events, vbucket DCP streams.
- **Settings:**
 - dcp_stream_boundary - From where the dcp stream should start from. Supported configs:
 - *everything* - Start from seq number 0 till end seq 0xFFFFFFFF
 - *from_now* - Start from current vbucket seq numbers till end seq 0xFFFFFFFF
 - worker_count - Number of V8 worker instances to spawn per eventing node for a specific event handler.

- log_level - Logging level of eventing related processes. Could be INFO, ERROR, WARNING, DEBUG, TRACE level - default being INFO
- function_timeout - Maximum duration(in ms), for execution of handler function in Javascript against a single document.

BETA BUILD Activity Documentation Support Administrator ▾

Eventing IMPORT CREATE

Dashboard
Servers
Buckets
Indexes
Eventing
Search
Query
XDCR
Security
Settings
Logs

Name

Description

RBAC Credentials

RBAC username

RBAC password

RBAC role

▼ Settings

DCP stream boundary

Log level

► Timer Settings

Cancel Save

Deployment Configuration

```
1 { "buckets": [{ "alias": "credit_bucket", "bucket_name": "credit_score" }], "metadata_bucket": "eventing", "source_bucket": "default" }
```

⊞

Copyright © 2017 Couchbase, Inc. All rights reserved.

Fig 4: Settings for “credit_score” Event Handler

Rebalance

As mentioned under [features](#), Eventing service will support MDS i.e. end-users could add nodes with eventing role and event processing throughput will scale up horizontally. Hence the eventing service provides facility to add/remove eventing service node(s) from the existing cluster.

There are 3 different cases that the eventing service needs to handle:

- Addition/Removal of eventing nodes

- Gracefully handle Key-Value service node(s) rebalance because eventing process is listening to source_bucket for events - hence it should follow vbuckets which are moving across different KV nodes during KV rebalance.
- Each eventing node has multiple V8 worker processes running. It needs to make sure each worker instance is taking up equivalent amount of load(i.e. Number of vbucket streams).

Example of vbucket stream distribution in different cluster setups:

One Eventing(EN1) and one KV(KV1) node, and adding one more eventing node(EN2) to it. Assuming single event handler(“credit_score”) is deployed with 3 as “worker_count” i.e. W0, W1 and W2.

Before Rebalance	After Rebalance
KV1 - vb0 - vb1023	KV1 - vb0 - vb1023
EN1: W0: vb0 - vb341 W1: vb342 - vb682 W2: vb683 - vb1023	EN1: W0: vb0 - vb170 W1: vb171 - vb341 W2: vb342 - vb511 EN2: W0: vb512 - vb682 W1: vb683 - vb853 W2: vb854 - vb1023

Fig 5: Vbucket stream distribution across eventing nodes and corresponding V8 workers

From above example it can be seen each eventing node is handling 512 vbuckets each and each V8 Worker on each node is roughly handling same number of vbucket streams 171/170 each.

Topology awareness

Eventing implements the *service.Manager* interface from *cbauth_service*, as directed by ns_server - which few standard APIs that Eventing Service nodes need to follow.

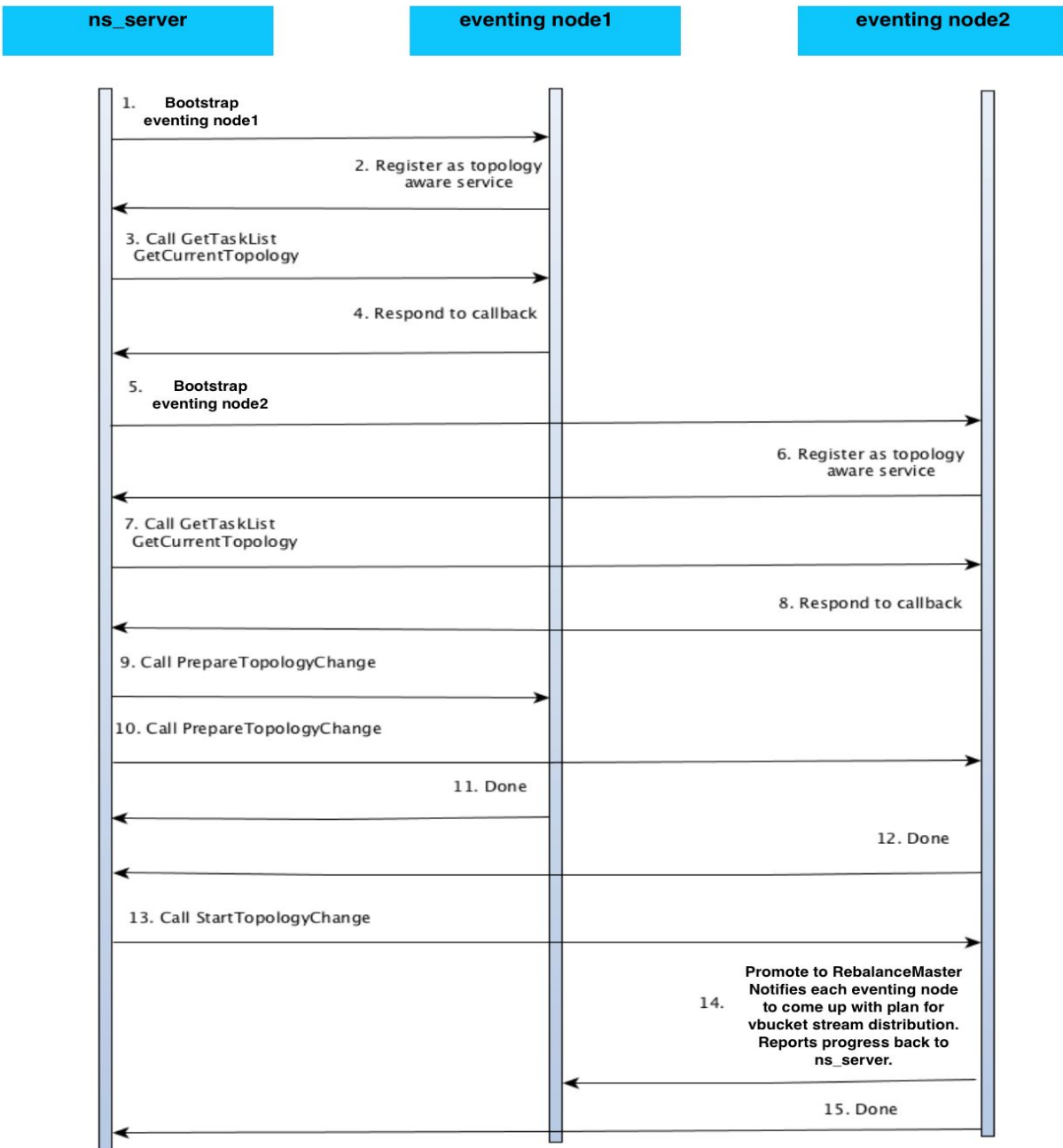


Fig 6: Eventing Topology Awareness Flow Diagram

Eventing Rebalance

Rebalance Can be divided into 3 phases:

1. **Prepare Phase** - ns_server supplies list of KeepNodes and EjectNodes to all Eventing nodes. Fig 7, shows format of message supplied by ns_server. ns_server waits for all nodes to respond *done* to PrepareTopologyChange call.
2. **Start Phase** - ns_server nominates one eventing node as RebalanceMaster(as per ns_server protocol definition, there could be more than one chosen master node). RebalanceMaster generates a unique RebalanceToken(same as rebalance changeId, supplied by ns_server), persists in MetaKV and each eventing node has subscribed to change in MetaKV for that path.
3. **Execution Phase** - On notification from MetaKV, each eventing node in the cluster will come up with deterministic plan for vbucket distribution across each eventing node and also break it down to V8 worker level.

```
type TopologyChange struct {  
    ID string `json:"id"`  
    CurrentTopologyRev Revision `json:"currentTopologyRev"`  
    Type TopologyChangeType `json:"type"`  
    KeepNodes []struct {  
        NodeInfo NodeInfo `json:"nodeInfo"`  
        RecoveryType RecoveryType `json:"recoveryType"`  
    } `json:"keepNodes"`  
    EjectNodes []NodeInfo `json:"ejectNodes"`  
}
```

Fig 7: Topology Change message structure sent from ns_server during Eventing rebalance

Both the Prepare and Start request needs to be converted to a predefined Task structure and ns_server keeps polling the status for the task using **GetTaskList** api. Fig 8 shows the defined message structure for task status reporting.

```

type Task struct {
    Rev Revision `json:"rev"`
    ID string `json:"id"`
    Type TaskType `json:"type"`
    Status TaskStatus `json:"status"`
    IsCancelable bool `json:"isCancelable"`
    Progress float64 `json:"progress"`
    DetailedProgressmap[NodeID] float64 `json:"detailedProgress,omitempty"`
    Description string `json:"description,omitempty"`
    ErrorMessage string `json:"errorMessage,omitempty"`
    Extra map[string] interface{} `json:"extra"`
}

```

Fig 8: Task status reporting to ns_server

If rebalance is stopped by the user, ns_server cancels the task using **CancelTask** api. This can happen at both the Prepare or Start stage. But in either case, **the new node is already part of the cluster and any node going out also remains as part of the cluster.**

Note:

As per *cbauth service.Manager* protocol, eventing service gets *PrepareTopologyChange* and *StartTopologyChange* only after KV/Views rebalance finishes because, eventing nodes listen on vbucket streams from “source_bucket” for a given event handler - during KV rebalance they would get DCP_STREAMEND from KV. For, vbucket streams that the eventing V8 worker should own(as per the deterministic planner) - it will try to respawn DCP stream(s) for those vbuckets first - before proceeding to apply change post *StartTopologyChange* call received from ns_server.

Prepare Phase

- On eventing node bootstrap, Rebalancer registers with ns_server as a topology aware service.
- When user hits rebalance, **PrepareTopologyChange** request is sent to **ALL** eventing nodes by ns_server, alongwith a list node UUIDs marked as KeepNodes and EjectNodes.
- Each eventing node will update list of nodes that it should use, to come up with vbucket DCP stream distribution map across eventing nodes and V8 workers running on them.

Start Phase

- On **StartTopologyChange**(only 1 node gets it and is promoted to RebalanceMaster), ns_server sends the list of nodes “wanted” and “ejected” in the cluster. Any node in the cluster can get this call including the one going out. See *pick_leader* in *service_rebalancer.erl*.
- RebalanceMaster will check if there is any RebalanceToken or TransferToken in MetaKV. If there are any, all tokens are deleted.

- RebalanceMaster will generate a unique **RebalanceToken** - which is same as ChangeId supplied by ns_server and persist it locally. The same tuple is persisted to MetaKV as well.
- All eventing have subscribed to changes in path where RebalanceMaster has written RebalanceToken. Hence they will get notified about of the change, whenever StartTopologyChange is received by RebalanceMaster.
- Upon getting notification from MetaKV, each eventing node will kick off planner to come up with plan for distribution of vbucket streams across all eventing nodes(KeepNodes is supplied by ns_server during PrepareTopologyChange Phase).
- Then eventing nodes will try to start vbucket DCP streams(if they haven't already) as per the vbucket DCP stream plan. If some other nodes has already opened up DCP stream for some vbuckets - it will wait for them to stop before opening. This guarantees that one and only eventing V8 worker has DCP stream open for a given "source_bucket".

Operations prohibited during eventing rebalance

- New event handlers can't be deployed while eventing rebalance is in progress
- Existing event handler(s) can't be deleted while eventing rebalance is in progress
- Event handler related settings like "source_bucket", "metadata_bucket" and "worker_count" can't be changed while eventing rebalance is in progress.

Writing event handlers

The event handlers are written in JavaScript. We expose the [N1qlQuery API](#) to carry out the database calls. We also provide syntactic sugar to write the query more naturally in JavaScript. Refer [transpilation](#) for description.

Timer Events

Assurance

Timer events will be fired at least once i.e. if a user has created a timer event, then eventing engine will fire it(even in case of KV rollback or KV data loss within metadata bucket - where we store timer related metadata)

credit_score::vb_100 (checkpoint metadata blob - CMB in short)

```
{
  "current_vb_owner": "10.1.1.1:25000" // eventing node owning dcp feed for vb 100 for credit_score
  "assigned_worker": "credit_score_worker_0",
  "last_dcp_seq_no": 100,
  "last_checkpoint_time": "2017-02-08T10:00:03AM+5:30",
  "vb_uuid": "abcdefgh12345",
```

```

    "timer_checkpoint_info": {
        "2017-03-08T10:00:00": "doc01", // Currently processing TMB and last processed doc_id
    },
    "timer_directory": [ // Ordered list of timer events to kick off in chronological order
        "2017-03-08T10:00:01": {"start_dcp_seq": 10, "end_dcp_seq": 20000},
        "2017-03-08T10:00:02": {"start_dcp_seq": 2, "end_dcp_seq": 18999},
        "2017-03-08T10:00:10": {"start_dcp_seq": 190, "end_dcp_seq": 599},
    ],
    // Some additional metadata
}

```

credit_score::vb_100::2017-03-08T10:00:00 (timer metadata blob - TMB in short)

```

{
    "doc01:cb01:10;doc02:cb02:20;doc03:cb03:21;....;doc1000:cb1000:20000",
}

```

Content of each entry is of format:

<doc_id>:<callback_func>:<dcp_seq_no_of_doc_id>

Sample entry:

docid1:updateUserCreditScore:10

Failure cases of timers

1. TMB rollbacks
2. CMB rollbacks
3. TMB is lost because of KV node failover
4. CMB is lost because of KV node failover
5. Wall clock time skewed on eventing node(s)
6. KV auto-failover is turned off, so vbuckets containing information about timer events to trigger are offline

Failure Handling of timers

- TMB rollback

Sample TMB blob has following structure

```

credit_score::vb_100::2017-03-08T10:00:00
{
    "doc01:cb01:10;doc02:cb02:20;doc03:cb03:21;doc1000:cb1000:20000",
}

```

It contains doc01, doc02, doc03, doc1000 and for those doc ids callback functions are cb01, cb02, cb03, cb1000 respectively. For all these doc_ids, timer events are supposed to fire at 2017-03-08T10:00:00.

But let's say above TMB gets rollbacked at 2017-03-08T09:00:00 to following state:

```
{  
    "doc01:cb01:10;doc02:cb02:20;doc03:cb03:21",  
}
```

So doc1000 related timer event metadata has gone missing from TMB. But we intend to provide at-most once model i.e. if a user has created a timer blob, we should fire it.

So, at 2017-03-08T10:00:00 i.e. when above TMB related docs need to be triggered - we would leverage seqno information stored in "timer_directory" field to figure out the start and end dcp seq nos which need to be present in 2017-03-08T10:00:00 TMB i.e. from 10 to 20000.

From TMB, we could see it only has seq no from 10 - 21, hence we will re-request dcp seq nos from 22 - 20000(which is the end_seq_no specified in CMB). Hence we will be able to restore TMB to original state prior to rollback

```
credit_score::vb_100  
{  
    "current_vb_owner": "10.1.1.1:25000",  
    "assigned_worker": "credit_score_worker_0",  
    "last_dcp_seq_no": 100,  
    "last_checkpoint_time": "2017-02-08T10:00:03AM+5:30",  
    "vb_uuid": "abcdefgh12345",  
    "timer_checkpoint_info": {  
        "2017-03-08T10:00:00": "doc01  
    },  
    "timer_directory": [  
        "2017-03-08T10:00:01": {"start_dcp_seq": 10, "end_dcp_seq": 20000},  
        "2017-03-08T10:00:02": {"start_dcp_seq": 2, "end_dcp_seq": 18999},  
        "2017-03-08T10:00:10": {"start_dcp_seq": 190, "end_dcp_seq": 599},  
    ],  
    // Some additional metadata  
}
```

CMB rollback

In this case we will start dcp streams from "last_dcp_seq_no" specified in rollbacked CMB. Given that we can't get notification on rollback event within metadata bucket unless we open up DCP stream from it - we could compare last snapshot of CMB(which eventing keeps in-memory) vs what's present in the metadata bucket at the time of checkpoint update(within metadata bucket).

TMB blob is lost

Like in the case of TMB rollback, we would restream dcp seqnos from “start_dcp_seq” to “end_dcp_seq”. That would recreate missing TMB.

CMB is lost

We would start dcp feed from the vbucket for which CMB has been lost from seq no 0

Wall clock time is skewed

We don't depend on wall clock to figure the next timer to be fired. Instead we depend on “timer_directory” within CMB, which is ordered according to time to fire.

KV auto-failover is turned off

Given KV auto-failover is turned-off, so in case of KV failover - vbuckets containing information about timer events to trigger are offline. Eventing engine will wait until offline vbuckets are back online. This case could happen even when more than KV nodes go offline at same time, even though auto-failover is turned on - cluster manager will wait for manual intervention.

Special considerations for Timer Events

- On redeploy of event handlers, it's needed to flush all previous TMB blobs which were created.
- For a given vbucket, only one timer processing routine(out of pool of worker routines) should run at any given time. This would avoid CAS retries against the same CMB.
- When cluster state changes, even though there are more timed events to process - current vbucket owner needs to given up ownership and before that it should checkpoint information about doc_ids it has already processed timed events.
- Additional opaque field need can be optionally allowed at the time of creating timer event. Using it user code could decide if they would want to skip processing of timer event for a specific doc id for whatever reason(maybe timer event is stale and no more needed)

ns_server integration

The integration with ns_server documentation on the protocol for topology aware services is based on the below two specifications:

TCMP1:

<https://docs.google.com/document/d/1VfCGOpALrqlEv8PR7KkMdnM038AOip2BmhUaoFwGZfM/edit#>

TCMP2:

Transpilation

We provide syntax sugar for the users to directly write the N1QL query in JavaScript as shown below.

```
function filter() {  
  var bucket = "beer-sample";  
  var res = SELECT name FROM :bucket WHERE type == "brewery" LIMIT 5;  
  
  for(var item of res) {  
    console.log(item);  
  };  
}
```



We also provide a memory-efficient iterator based on the [N1qlQuery API](#) for iterating through the results of a database query.

Transpilation features

Using N1QL as part of JavaScript

The transpiler enables the user to write the N1QL query “as-is” in the host language, as shown above.

Benefits:

- The query will now look more elegant and natural.
- Improves readability.

Variable substitution

Any variable that needs to be substituted in the query must begin with a colon ‘:’. The transpiler will determine the variables that need to be substituted in the N1QL query and automate the process of substituting them in the query string, subsequently followed by making a call to the database. For example, referring to Figure (1) the variable in the query string :bucket will be substituted by the variables declared in the JavaScript code, at runtime.

Benefits:

- The user gets to decide which variable gets substituted in the query.
- An elegant way of writing the query, as opposed to fragmenting the query string to substitute parameters.

Examples:

```

function update () {
  var type= "actor";
  var res = UPDATE tutorial USE KEYS "baldwin" SET type = :type RETURNING type;
  for(var item of res) {
    console.log(item);
  });
}

function join(bucket ) {
  var key = "Tamekia_13483660";
  var res =
    SELECT usr.personal_details, orders
    FROM :bucket usr
    USE KEYS :key
    LEFT JOIN orders_with_users orders ON KEYS
    ARRAY s.order_id FOR s IN usr.shipped_order_history END;
  for(var item of res) {
    console.log(item);
  });
}

function insert() {
  var bucket = "travel-sample";
  var value = { "id": "01", "type": "airline"};
  var res = INSERT INTO :bucket ( KEY, VALUE )
    VALUES ( "k001", :value)
    RETURNING META().id as docid, *;
  for(var item of res) {
    console.log(item);
  });
}

```

Iterator

The iterator construct is in accordance with the ECMAScript standard. If we detect that the source of the iterator is the result of a call to database, we stream the result item - by - item, for every iteration of the loop. For example, the the item is fetched at the beginning of the iteration, the next item is fetched only when the next iteration begins. All items are not fetched at once. Breaking out of the iterator will stop the streaming of results. For more details, refer to [Streaming cancellation](#) under the [Optimizations](#) section.

Benefits:

- It is memory friendly - All rows need not be loaded into memory at once.
- Partial processing - One of the scenarios where iterators are useful, is for partial processing of results. For example, a user may want to stop iterating after the 100th row out of 1 million rows.

Hence, the iterator will fetch only the first 100 rows, as opposed to loading all the 1 million rows into memory.

N1QL Query API

We are exposing an API for JavaScript to carry out the database activities for querying and retrieval.

Constructor

N1qlQuery(string[, options])	<p>Arguments</p> <ol style="list-style-type: none">1) N1QL statement as a string. This could be a 'tag template literal', as supported by ES6.2) options is a JSON - {prepare : boolean}.<ol style="list-style-type: none">a) Setting 'prepare' to true will treat the query as a prepare statement which will be useful if query is run multiple times. <p>Returns an instance of N1qlQuery.</p>
------------------------------	--

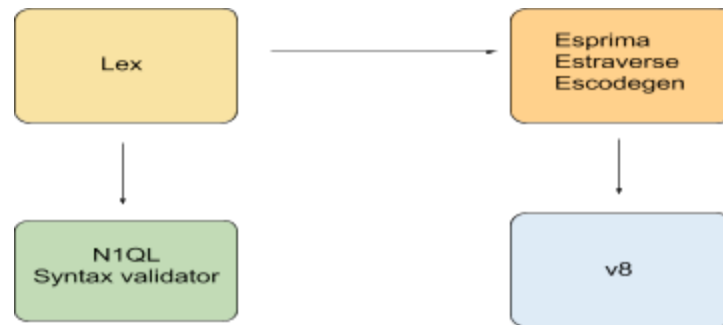
Properties

query	It is the query which will be sent to the server upon calling execQuery() or iter().
-------	--

Instance methods

execQuery()	Blocks execution and sends the query to the server and waits till all the results are loaded into memory. It returns a list of the result.
iter(callback)	<p>Arguments</p> <p>Accepts a callback with parameter. Executes this callback synchronously as each row arrives, passing the row as the argument to the callback.</p> <p>Returns 0 if a call was made to stopIter ().</p>
stopIter()	Stops streaming of rows and hence will prevent further execution of callbacks. However, it will not stop the execution of the current iteration.

Transpilation components



Transpiling N1QL to JavaScript involves the following stages:

1. Pre-processing
 - a. Phase 1
 - b. Phase 2
2. Transpilation
3. Execution

Pre-processing

This stage mainly involves validation of user input followed by getting an intermediate representation of the N1QL statements embedded in the JavaScript program. It involves two phases (1 and 2) as described below.

Consider the following input,

```
function OnUpdate() {  
    var bucket = 'default';  
    var res =  
        SELECT *  
        FROM :bucket ;  
    for (var row of res) {  
        log (row);  
    }  
}
```

An example of handler code

Phase - 1

Most of the validation of user input takes place in this phase. The workflow is:

1. The Lex component will identify the N1QL statements in the JavaScript program and passes these to a N1QL syntax validator. After it passes the N1QL syntax validation, it will comment these statements (necessary for next step).
2. It will then pass this code to esprima component which will validate the JavaScript syntax.
3. Once the JavaScript syntax has been validated, we then check if identifiers are valid.
4. We check to ensure anonymous functions are not used.

If any error occurs in any of the four steps described above, an appropriate error code is returned to the go routine invoking this component.

The output of Phase - 1 is discarded as it is unnecessary to roll back the transformations that were done as part of validation.

Phase - 2

Since the input has been processed through Phase - 1, we now know that all the N1QL statements are syntactically valid. The embedded N1QL statements are transformed to an instance of [N1qlQuery](#) class.

The output of Phase - 2 is as shown below:

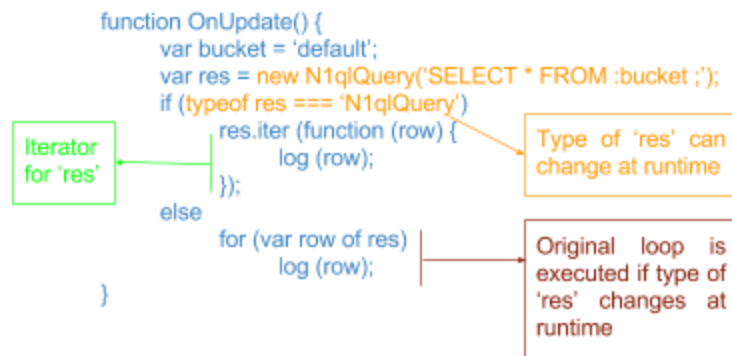
```
function OnUpdate() {  
  var bucket = 'default';  
  var res = new N1qlQuery('SELECT * FROM :bucket ;');  
  
  for (var row of res) {  
    log (row);  
  }  
}
```

Transformation of embedded N1QL statement

Transpilation Stage

In this stage, transformations are applied to the usages of N1qlQuery instances (in case of iterators). We first perform checks to detect and validate N1QL syntax and variable substitutions correctness. Appropriate error codes are returned if the above constraints are not satisfied.

We now traverse the Abstract Syntax Tree (AST) of the JavaScript program and apply transformation to the 'for ... of ...' constructs. Please refer to [iterator mechanism](#) section for more details on how the transformation occurs. The output of this stage is shown below,



Transpilation output

A quick check is performed to determine the nature of the query. If the query is not SELECT, we call `execQuery ()` right after instantiation of `N1qlQuery` as shown below.

```

function OnUpdate() {
  var bucket = 'default';
  var res = (new N1qlQuery ('DELETE * FROM :bucket;')).execQuery ();
  for (var row of res) {
    log (row);
  }
}

```

Transpilation output

Execution Stage

During runtime of the program, just before the query is sent, variable substitution (substitute values for `<var>` N1QL statement) happens. It is essential to do the variable substitution at runtime because JavaScript is a dynamically typed language and a variable can get defined at runtime and its type could also change at runtime.

It is to be noted that, we're performing a type-check at runtime, if the type of the `N1qlQuery` instance changes, query is not sent and hence the iterator won't run, the program will fallback to the original loop that the user wrote.

Iterator mechanism

The iterator support is provided only for the queries involving SELECT. As depicted in the sections above, the 'for ... of ...' construct only serves as a syntax sugar. Under the hood, the 'for ... of ...' constructs get transformed into calls to `iter()` method - only if it finds the source of the 'for ... of ...' construct is an instance of `N1qlQuery`. The crux of the transformation of 'for ... of ...' construct into

iterator is to bridge the semantics of the loop (the ‘for ... of ...’ construct) with the callback (the iter() method).

The bridging of the semantics of ‘for ... of ...’ loop with callback implies that we need to handle to entry to and exit from the ‘for ... of ...’ loop, meaning, we must impart the same behaviour to the callback.

The main challenge here is that the callback introduces a new scope and thereby changes the semantics whereas the ‘for ... of ...’ loop does not. Thus, in order to impart the behaviour of the ‘for ... of ...’ loop we need to match the scope of the callback with that of its parent and also match the semantics.

Entry criteria

The entry to the iterator is dependent on whether the source is an instance of N1qlQuery. This is done by checking its type just before the iterator.

Exit criteria

This primarily deals with those constructs that can modify the behaviour of the ‘for ... of ...’ loop, listed below:

1. Labelled and unlabelled break and continue
2. return
3. throw
4. The association of the above statements with the loop

Using the AST, we detect whether the above statements can cause a change to the iterator loop’s iteration and the level of nesting upto which it affects. We modify only those statements that modify the loop’s behaviour.

The following examples show the modifications that take place. It is to be noted that the following only serve to illustrate the modifications but do not enumerate all the possible cases.

In the following examples,

1. Type check is omitted for brevity.
2. ‘res’ is a N1qlQuery instance.

Unlabelled break

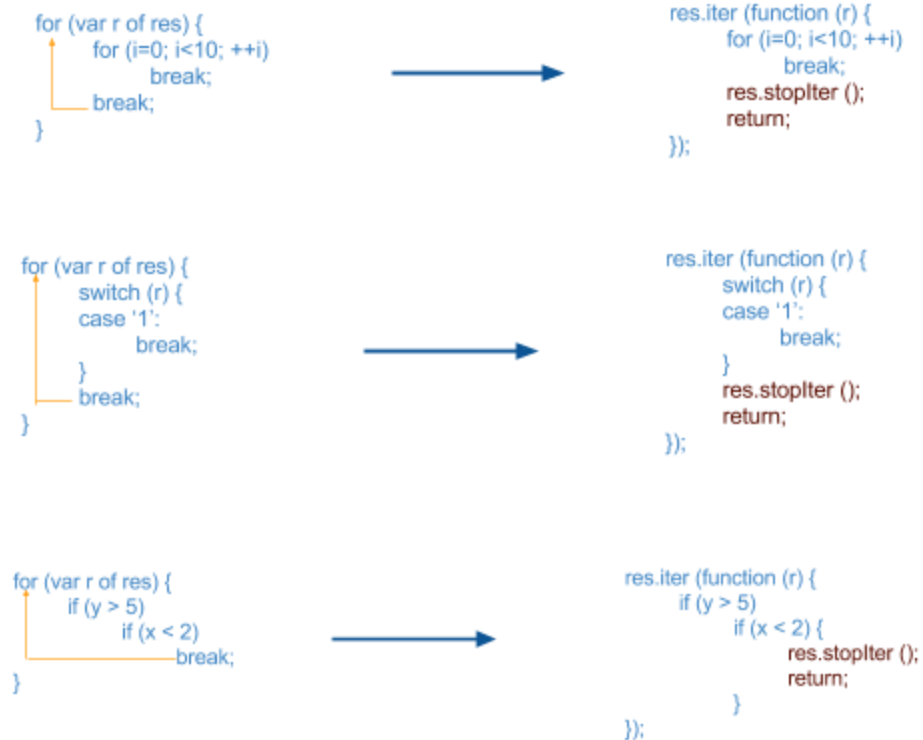
‘break’ translates to stopIter() followed by a return.

The diagram illustrates the translation of a JavaScript 'for' loop with a 'break' statement into a callback-based iteration. On the left, a 'for (var r of res) { log (r); break; }' loop is shown. An orange bracket highlights the 'break' statement. A blue arrow points to the right, where the equivalent code is shown: 'res.iter (function (r) { log (r); res.stopIter (); return; });'. The 'res.stopIter ();' and 'return;' lines are highlighted in red to show they are the result of the translation.

```
for (var r of res) {  
  log (r);  
  break;  
}
```

→

```
res.iter (function (r) {  
  log (r);  
  res.stopIter ();  
  return;  
});
```

Unlabelled continue

The transformation to 'continue' statements is similar to that of 'break' statement shown above, except that 'continue' gets replaced by 'return'.



Labelled break



Labelled continue

```
outer :  
for (var r of res) {  
  for (i=0; i<10; ++i)  
    break;  
  for (j=0; j<10; ++j)  
    continue outer;  
}  
}
```

→

```
outer : {  
  res.iter (function (r) {  
    for (i=0; i<10; ++i)  
      break;  
    for (j=0; j<10; ++j) {  
      return ;  
    }  
  });  
}
```

Return

The iter() function will return 0 if a call was made to stopIter(). Upon returning from iter () function

```
function OnUpdate () {  
  for (var r of res)  
    return;  
}
```

→

```
function OnUpdate () {  
  res.code = res.iter (function (r) {  
    res.stopIter ();  
    return;  
  });  
  if (res.code == 0)  
    return;  
}
```

Throw

In case of a throw statement, we call stopIter() in all the catch blocks associated with the throw statement.

```
function OnUpdate () {  
  try {  
    for (var r of res)  
      throw 'Error';  
  } catch (e) {  
    log (e);  
  }  
}
```

→

```
function OnUpdate () {  
  try {  
    res.code = res.iter (function (r) {  
      throw 'Error';  
    });  
  } catch (e) {  
    res.stopIter ();  
    log (e);  
  }  
}
```

Uncaught exceptions

This is handled by v8. The callback to iter () is executed within a global try ... catch in v8. During the execution of the callback, if the program ends abnormally (like zero division error, uncaught throw statements), the exception is caught by the global catch block and streaming is stopped. The exception then is re-thrown by the global catch block.

Nested iterators

This typically attributes to the scenario of iterating through a N1qlQuery instance when already in the N1qlQuery iterator.

```

for(var r1 of res1)
  for(var r2 of res2)
    log(r1, r2);

```

Bubbling

When inside a nested iterator, if there is a labelled break which breaks from the innermost loop out to the outermost loop, the `stopIter()` has to be called on all the loops that are nested in between. Thus, we say that the `stopIter()` must be “bubbled” all the way to the target label. Another example would be, if ‘return’ is called from the innermost loop, `stopIter()` must be called on all the enclosing loops before exiting the function.

Bubbling design

As shown in the [transpilation](#) section, the for-of loops undergo transformation to an if-else block to enforce type checking. We capture how the iterator will exit into the iterator variable itself (something like ‘res.x’ where ‘res’ is a N1qlQuery instance). This prevents us from polluting the namespace.

```

x: while(true)
  for(var r of res)
    break x;

```

→

```

x: while(true)
  if(typeof res === N1qlQuery)
    res.x = res.iter(function(r) {
      return res.stopIter('breakx');
    });
  else
    for(var r of res)
      break x;

```

Now we know how the iterator exited (could be labelled break/continue, return etc). We now add a switch-case block following the iterator. Depending on how the loop exited, the same response will be ‘bubbled-up’ to the loop above -

```

x: for(var r1 of res1)
  for(var r2 of res2)
    break x;

```

→

```

x: // inside res1 iterator
  if(typeof res2 === N1qlQuery) {
    res2.x = res2.iter(function(r) {
      return res2.stopIter('breakx');
    });
    switch(res2.x) {
      case 'breakx':
        res1.stopIterator('breakx');
    }
  }
// else block

```

Internally, how it works is that, the nodes that have to be switched (from 'stopIter()' to 'break' and vice versa) will be annotated as special nodes. During traversal of the AST, when the transpiler comes across these annotated nodes, it makes a check to verify the reachability of the statement. If it's not reachable, it calls stopIter() it will retain the original statement otherwise.

Optimizations

Optional query execution

This optimization applies only for the queries involving SELECT. Consider the following scenario.

```
function update () {  
  var type= "actor";  
  var res = SELECT name FROM artists WHERE type = :type;  
  return 'done';  
}
```


In the above code, the result of the query is stored in the variable res. But res isn't used anywhere in the rest of the code following its declaration. Since the result of the query isn't used, it makes sense to not execute the query at all and hence, we can drop such queries.

It is to be noted that this optimization isn't applicable to other operators like INSERT, DELETE, UPDATE etc. which involve writing/modifying the database because it would lead to inconsistency.

Streaming cancellation

The logic in the iterator may involve a premature exit, without going through all the elements as shown in the example below.

```
function OnDelete (state) {  
  var bucket = "beer-sample";  
  var res = SELECT name FROM :bucket WHERE type == "brewery" LIMIT 5;  
  
  var count = 0;  
  for (var item of res) {  
    if(count == 5) |  
      break; |  
    console.log(item);  
    ++count;  
  });  
}
```



As shown above, the iterator will stop after running through 5 elements of the result. In such cases, the iterator will stop streaming the results and dispose the handle so that it doesn't hold back any of the resources.

Constraints

Disallow conflicting identifiers

The users will not be allowed to use the words SELECT, UPDATE, DELETE, UPSERT, INSERT, MERGE to be used as identifiers for variables or function names or label names.

Reason - The Lex component identifies the start of a N1QL statement with the above words. However, these words can appear as part of comment or a string, as they are ignored.

Disallow anonymous functions

The users are not allowed to have anonymous functions in any part of the program. This is required so eventing does not have to maintain a long lived node local state of execution.

N1QL statement declaration

The N1QL statements must be assigned to a variable or it must be a statement. It can't, for example, appear as a parameter to some function call etc. This is necessary to prevent undefined behaviour.

For example,

```
var res1 = SELECT * FROM default ; ✓  
DELETE beer FROM `beer-sample` ; ✓  
log (SELECT * FROM `beer-sample`); ✗
```

Variable substitution in N1QL statements

We allow only variable substitution in the embedded N1QL statement and the variable must be active at runtime for substitution. For example,

```
var bucket = 'default';  
var res = SELECT * FROM :bucket; ✓  
DELETE beer FROM :getBucket (); ✗
```

Failure Behavior

Hard failover of eventing node(s)

This is the expected behavior when a eventing node just dies or ns_server does a hard failover of an eventing node (causing it to lose access to all cluster secured resources):

1. Until rebalance is triggered, doc_id based timers mapping to vbuckets assigned to that node will not be fired.
2. When rebalance is triggered, backfill from 0 will be initiated from new eventing node(s) who are new owners of those vbuckets. This is to recreate the node local doc timer cache, and will not retrigger handlers.
3. Any timer event(s), which was/were supposed to kick off between the window of hard failover and rebalance will be delayed in their execution. Magnitude of delay would vary.
4. Handlers that had executed but checkpoint was not updated will fire again on some other node. The amount of duplicate timers will be bounded by checkpoint interval (proposed time: 25 seconds)
5. Doc timers that had executed but checkpoint was not updated will fire again on some other node. The amount of duplication is bounded by the checkpoint time.
6. Non doc timers are stored in timer groups which are also checkpointed at the checkpoint interval. When node hard fails over, there will be a duplication firing of non doc timers limited to the maximum window size of checkpoint interval.

Hard failover of KV node(s) with KV replicas

- Until rebalance is triggered, all non_doc_id based timers whose metadata was in the failed vbuckets will stop processing / triggering. At present, Eventing doesn't read from vbucket replicas.
- Rollback in metadata bucket. If timer blob is rolledback:
 - If non_doc_ids timer blobs are rolled back, Eventing will retrigger all the timers that are indicated as needing execution by the rolled back metadata. This behavior is bounded to the timers created in the interval the data rolled back by plus checkpoint interval.
- Rollback in metadata bucket. If dcp checkpoint blob is rollbacked, 2 cases:
 - If vbucket STREAMEND has been received for one such vbucket, then it will start from start_seq # in rollbacked dcp checkpoint blob
 - Else it will keep going with dcp stream processing and update checkpoint blob at checkpoint interval.

- dcp checkpoint blob and non_doc_id timer metadata blob, both rollback to different snapshots. Then eventing will simply respect the start seq # in dcp checkpoint blob.
- If replica vbuckets aren't caught up with active vbuckets and also behind eventing process, in terms of seq number processed so far. Then Eventing process will rollback to seq number that replica node has due to vbuidid branch seen.
- In general, if KV data and eventing metadata both rollback, eventing will respect the eventing metadata worldview, until it actually disagrees with KV data (such as vbucket branch being observed).
- Operations that will kick off only post rebalance:
 - STREAMREQ for vbuckets from which STREAMEND was received.

Differentiating CE vs EE

In general, we want CE to be featured enough to effectively replace View Engine programmability use cases. And where we derive significantly from external open source code, we want to keep it in CE. Other features tend to gravitate towards EE.

Further discussion tracked on source document, [Function Service - CE vs EE](#)

Features	CE	EE	Closed Source
Architecture & Scaling			
Multi-Dimensional Scaling	No	Yes	
Message Delivery Guarantees			Yes
Development Tools			
Javascript Editor	Yes	Yes	
Embedded Javascript Debugger	No	Yes	
Lambda Log Management	Yes	Yes	
Lambda Monitoring	Yes	Yes	
Pausing Lambda's event processing	No	Yes	

(Disable/Enable)			
Functional Capabilities			
N1QL Queries in Event Handlers	Yes	Yes	
Document Timer Processing	No	Yes	Yes
Non Document Timers Processing	No	Yes	Yes
Recursive Mutation Handling	Yes	Yes	
Multiple DCP stream boundaries	No	Yes	
Skipping delayed timer events		Yes	
Support for cleaning up of timers on Function redeployment	Yes	Yes	
Integrations			
Ability to make CURL calls inside the Event Handler	No	Yes	Yes
JDBC/ODBC connectivity	No	Yes	Yes
HTTP/S Invocation	No	Yes	Yes
Security			
RBAC support	No	Yes	
Performance			
Tunable V8 worker processes	No	Yes	
Batched messaging between Go and C++	No	Yes	

Future			
Predicate Based Streams			
Dependency Resolution (for Cloud)			

Dependencies

Internal dependencies:

- ns_server for managing MDS and cluster membership, and providing topology aware container for eventing
- ns_server for hosting the pluggable UI
- MetaKV is used for storing lambda handler code and their respective settings (like workerCount, cleanupTimer, dcpStreamBoundary, logLevel and more). This is basically the eventing design time artifacts. All runtime artifacts are in KV (either in bucket or in XATTRs).
- MetaKV is used for storing rebalance Token

Third Party Dependencies:

- Go
- V8
- Libuv
- Flatbuffers
- Flex, Esprima, Escodegen, Estraverse, SourceMap
- Ace editor

Open questions:

Framework

- Expose flag at lambda level to once and only once model for dcp events at the expense of missing out on timer events?
- For doc_id having timer associated with it(assuming it's to notify about doc expiry), should eventing avoid triggering of OnDelete handler?

Editor/UI

- From UI, against a lambda 4 management options are possible - Enable/Disable/Delete/Save.
- There will be max 2 versions of a specific lambda, that eventing will maintain. First is the deployed version and second is staging version that the user is working on and just saving it - later on user may or may not choose to deploy that version of handler.
- UI will allow to import a new archive, containing lambda handlers code and deployment config(just zip archive for now). If settings data(worker count, checkpoint interval and others) are missing from archive, default values for them will be picked up. Imported lambda will by default be in undeployed state.
- UI will as well allow to expose export option against a deployed/undeployed lambdas, which would encapsulate handler code, deployment config and different settings for the lambda.

Upgrade

- RPC protocol used for rebalance for timer data transfer
- Vbucket distribution planner
- Plasma file format from older eventing node to newer node
- Format of checkpoint blobs stored in metadata bucket
- Format of timer related data written into xattr
- Format of timer related data written to plasma
- Cluster wide aggregate stats endpoints for reporting back dcp/timer event ops stats to UI. Also used during rebalance operations to get list of RPC tcp endpoints to fetch timer files.