SDK-RFC 73: KV Range Scan

Meta

- RFC Name: KV Range Scan
- RFC ID: 73
- Start-Date: 2022-06-27
- Owners:
    - David Nault
    - Michael Nitschinger 👻
- Current Status: DRAFT
- Revision #9

Motivation

With support for key-prefixed range scans in the data service, users do not need to provision index and query nodes if they do not have other query use cases. This can greatly reduce TCO and also allows discovering document IDs purely using the data service.

Overview

KV Range Scans are an addition to the data service enabling clients to request and retrieve documents from a single collection in persistent buckets using a key-range as input. This is the first API available to users where a range of keys can be loaded with one user-level operation. Before this feature, the only way to achieve this would be through Couchbase's various secondary indexing features.

A range scan lets a user iterate over existing document IDs from starting position up to an end position.

For example "`user-east:<id>`" signifies a user in a location with a unique ID. With this pattern other user's may exist in a `'north'`, `'west'` or `'south'` location.

Key Range Scans will enable a user to utilize those key patterns to retrieve subsets (inclusive or exclusive ranges) of documents from a single collection without the need to set up a DCP stream or use secondary indexes.

In this example the following patterns could be used to retrieve:

- All users with "`user-`"
- All users in the southern region with "`user-south`"
- All users in the northern region with "`user-north`"

User-Facing API

Since KV range scan operates on a collection, it is placed at the `Collection` level:

```
Collection {
  …
  Stream<ScanResult> scan(ScanType scanType, [ScanOptions options])
  …
}
```

The `scan` operation returns a `Stream` of `ScanResults`, where the Stream type itself differs per language (similar to `getAllReplicas`).

Since a KV range scan supports two modes of execution (scanning a range, and random sampling), a ScanType is introduced as a marker interface and then two different implementations are defined. Languages which do not support this type of extension should use alternative names for the scan method (i.e. `scanPrefix, scanRange`, and `scanSampling`).

```
interface ScanType {}

PrefixScan implements ScanType {
    String prefix
}

RangeScan implements ScanType {
    ScanTerm from (optional)
    ScanTerm to (optional)
}

SamplingScan implements ScanType {
    Uint64 ("json number") limit (required, > 0)
    Uint64 ("json number") seed (optional, default is random uint64)
}
```

CAUTION: The server rejects negative seed values. Make sure the value in the JSON is non-negative.

When performing a `RangeScan`, the from and to terms may be specified. Each term is composed of a `term` value as well as an `exclusive` flag which controls whether the term is inclusive or exclusive.

```
ScanTerm {
      String term()
      boolean exclusive()
}
```

Implementations should provide an idiomatic way to create a `ScanTerm`. If the `exclusive` option is not explicitly specified when creating the `ScanTerm`, it defaults to false.

When translating a `RangeScan` into a Memcached request, absent endpoints are encoded as follows:

- Absent `from` -> Inclusive ScanTerm with single character U+0000 (NUL)
- Absent `to` -> Exclusive ScanTerm with single character U+10FFFF (LAST)

For reference, code point U+0000 is represented in UTF-8 as the single byte 0x00. Code point U+10FFFF is represented in UTF-8 as the 4-byte sequence: 0xf48fbfbf.

When deciding whether a document ID falls within the requested range, the server does a bytewise comparison of the UTF-8 bytes of the ID. Collation (locale-specific rules for string sorting) is out of scope.

The “PrefixScan” scan type selects every document whose ID starts with a certain prefix. The start of this range is the given document ID prefix. The end of the range is derived by appending U+10FFFF to the prefix.

The `ScanOptions` are composed of the common options that every KV operation exposes as well as custom options specific to the Scan operation. See later section for specifics.

```
ScanOptions {
      // Common options
      Timeout
      Transcoder
      RetryStrategy
      ParentSpan

      // Specific Options
      idsOnly(boolean);
      consistentWith(MutationState);
      batchByteLimit(uint32);
      batchItemLimit(uint32);
      batchTimeLimit(Duration); // SEE NOTE
      concurrency(uint32); // SEE NOTE
}
```

The `batchByteLimit` specifies how many bytes are sent from the server to the client on each partition batch. The `batchItemLimit` is similar, but limits the number of documents or document IDs in each batch. Both batch limit options may be set at the same time. The server response completes when any one of the limits is met, or the end of the scan is reached.

NOTE: If the SDK implements timeouts by applying the timeout to each individual stream emission, then `batchTimeLimit` should be omitted from `ScanOptions` and should always be set to 90% of the client-side timeout (similar to how the client sets timeouts for KV durable writes).

NOTE: The `concurrency` option specifies how many vbuckets the client should scan in parallel. If the SDK does not support scanning vbuckets in parallel, this option should be omitted from `ScanOptions`.

Defaults that must be set by the SDK:

- **batchByteLimit**: `15000`
- **batchItemLimit**: `50`
- **batchTimeLimit** (if applicable): `0`
- **concurrency** (if applicable): `1`

If the operation is successful, a stream of `ScanResults` is returned, where each one looks and behaves very similar to a `GetResult` with some implementation differences that are discussed later (content and metadata might not be streamed).

```
ScanResult {
      String id();
      T ContentAs<T>(); (Optional, present only if idsOnly is true)
      PointInTime ExpiryTime(); (Optional, present only if idsOnly is
      true and document has expiry)
      Uint64 Cas(); (Optional, present only if idsOnly is true)
}
```

High-Level Architecture

The overall flow of a range scan interaction can be described as follows:

For each vbucket (sequentially or with parallelism if supported as defined below):

1. Client sends a “range scan create” request for one vbucket.
2. Client receives either a scan UUID, or a NOT_FOUND (0x01) status indicating the scan would return no items. If NOT_FOUND, skip the rest of these steps and move on to the next vbucket.
3. Client sends one “range scan continue” request to that vbucket, passing the scan UUID from step 2.
4. Client receives a batch of data which is returned in a streaming fashion to the user.
5. If the “range scan continue” status indicates the scan is complete, move on to the next vbucket. Otherwise, go to step 3.

For clients which support scan concurrency, the SDK must expose a “max_concurrency” option that defaults to 1 (or the number of nodes in the cluster, if node affinity is supported). The SDK must initially start with this number of vbucket scans occuring in parallel. The SDK should attempt to execute these requests with “node disaffinity”, distributing the scans across distinct nodes in the cluster (1 scan per node by default). The SDK must recognize the ‘EBusy’ status code returning from a RangeScanCreate on one of these parallel threads and stop that thread when this occurs (effectively reducing the level of concurrency of the scan). Note that the SDK should never scale down below a concurrency level of one; Ebusy on the last remaining thread must retry rather than stopping.

Clients which do not support scan concurrency must not expose any tunables related to concurrency.

All requests related to the same stream must be issued on the same KV connection. If the client pools KV connections, all requests for the same stream must be pinned to the same connection.

The key point is that the client is responsible for opening and managing “streams” for each vbucket on the cluster. All vbucket streams are logically merged together and returned as one “stream” to the user. Error handling is also performed on a vbucket stream basis and as a result the client needs to keep track of certain states along the way.

One important aspect is that the batch returned for a continuation request will potentially return multiple KV responses for the same request. This is similar to the “stats” command, but will likely require changes in logic in the underlying I/O layers in SDK 3 since up to this point KV returned a single response for a single request.

Implementation Details

KV Range Scan operations all operate at the Collection level, so there is a Bucket, Scope and Collection implicitly available for all operations mentioned below.

The server config includes a `bucketCapability` named “`rangeScan`” if present (both in 7.2 builds and 8.0 builds).

Protocol Flow

Range Scan Create

For every vbucket in the collection/bucket, a Create Range Scan command must be issued. The configuration for the range scan is sent as part of the request body (no key, no extras) as a JSON payload. See the server side spec linked above for all the possible options. The following payload illustrates an example request for a range scan:

```
{
  "collection": "f2",
  "range": {
    "end": "dXNlcv8=",
    "start": "dXNlcg=="
  }
}
```

A sampling payload looks like this:

```
{
  "collection": "f2",
  "sampling": {
    "seed": 18111,
    "samples": 512
  }
}
```

If all partitions returned with a successful result, each response will have a unique Scan UUID that needs to be passed to the following continue requests. Failure handling is discussed in a different section.

Note that one special case needs to be mentioned: if 0x01 is returned as a status (ENOENT), then the range is empty and an empty stream should be returned immediately.

Range Scan Continue

For each partition a Scan Continue Request must be sent. The extras of the request contains

- The Scan UUID for that partition

- Item Limit
- Time Limit
- Byte Limit

If no limits are set, 0 needs to be sent instead - the UUID is required.

The response is special. For one request potentially more than one response is returned, which means that the I/O layer of the SDK needs to add in special logic to handle this case *(i.e. in java we put the request back into the sent map with the opaque so that the next request can be handled again)*.
The responses are (presently) split by kv_engine at the boundary of value size of 8192 bytes.

The status of each individual response indicates the next steps:

- Success 0x00: Used for intermediate responses making up a larger response. The request needs to be "put back" so since more responses for the same opaque will arrive.
- RangeScanMore 0xA6: Scan has not reached the end key, more data may be available, client should issue another continue.
- RangeScanComplete 0xA7: Scan has reached the end of the range, no more continue needed.

The payload of each individual response depends on the option if only the IDs are returned or ID plus content. Both RangeScanMore and RangeScanComplete statuses can also contain a value that must be handled in the same way as Success status. See the KV engine design document for more details on the protocol spec.

Each individual item needs to be emitted into the stream (or the component assembling the streams). When the SDK has emitted a number of items equal to the user provided limit for a sampling scan, then no more items may be emitted into the stream. When this happens, all other streams related to the sampling scan should also be canceled; as below, this cancellation is "best effort."

Range Scan Cancellations

If a range scan continues until the end, there is no need to cancel it.

After a successful `range-scan-create` request the client must cooperatively cancel the range scan at any time. This will allow the server to release all resources associated with a scan. Failing to cancel range scans will result in the server holding the range scan and associated resources

When an individual "partition continue" fails or some other condition is hit which prevents the stream from completing (i.e. someone unsubscribes early from a stream), then all scans should be canceled to allow the server to clear up the bound resources quickly.

This is best effort though (since the user likely has already received a terminal result) and the server will clean up open streams internally after some time as well.

Note: In the future KV-Engine might introduce another command which allows to cancel all partitions in one shot per node, but right now it still needs to be done on a per vbucket basis.

KV Scan Timeout

In the data service, every scan operation will hit the disk (since the index used is maintained by couchstore or magma). As a result, the expected latency is much higher than for typical KV operations.

Similar to the higher timeouts like "Query Timeout", a new "KV Scan Timeout" is introduced which also defaults to 75 seconds (instead of the default 2.5s for a regular KV operation).

- `kvScanTimeout`(): Default **75s**

This timeout can be customized through the ScanOptions (the timeout option) like similar timeouts.

The scan fails with an `UnambiguousTimeout` error if the first item is not received before the timeout expires. (Or some variation on this theme.)

ConsistentWith Behavior

The `ConsistentWith` option takes a `MutationState` which is split into (potentially) many `MutationTokens`. Since a kv range scan also multiplexes multiple vbuckets, it is possible to send a `MutationToken` to a vbucket (based on the vbucket ID) and apply "Read your own write" consistency, very similar to the same behavior in Query. If a MutationState contains multiple tokens for a single vbucket with differing vb-uuids, an error should be returned.

Here is a sample encoded format:

```
{
  "collection": "f2",
  "range": {
    "end": "dXNlcv8=",
    "start": "dXNlcg=="
  },
  "snapshot_requirements": {
    "vb_uuid": "16627788222",
    "seqno": 1000,
    "timeout_ms": 75000,
  }
}
```

Note that only `vb_uuid` and seqno need to be set, other options like `seqno_exists` are not exposed by the SDKs (for now).

MutationToken	**snapshot_requirements**
`partitionUUID`	`vb_uuid`
`sequenceNumber`	`seqno`

In addition, the `timeout_ms` must be set to the timeout configured on the operation.

ScanResult

A `ScanResult` should be based on (but not extend from) the SDK's existing `GetResult`, with the addition of an `id` property (String) and an `idOnly()` method (returns boolean).

The `id` property is always present.

If `idOnly()` returns true, it indicates only the ID property is present. In other words, a `ScanResult` that came from a scan where `idsOnly` was set to true must return true from its `idOnly` method.

It is a programming error for the user to access the `content`, `CAS`, or `expiryTime` of a `ScanResult` whose `idOnly()` method returns true. If the user accesses these properties anyway, languages that can throw exceptions should throw `NoSuchElementException` or equivalent. In languages where throwing an exception is not possible (or would not be idiomatic), these properties should be populated with dummy values. Specifically, `content` should be empty, `cas` should be 0, and `expiryTime` should be 0 / absent / Instant.EPOCH / etc.

Error Handling & Retry Strategy

For non-sampling scans, all effort should be made to keep the range scan streams going until timeout or completion. This also includes restarting individual partition sub-streams if at all possible (based on the errors outlined below).

The permanent failure of any range scan substream causes the whole range scan to fail. If multiple concurrent substreams fail permanently for different reasons, the SDK arbitrarily selects a single failure to report to the user as the cause of the overall stream failure.

For sampling scans, failures that occur while processing a substream are not retried, and do not cause the overall stream to fail. The rationale is that sampling scans are “best effort” to begin with, and it’s not possible to resume from a specific point in a sampling scan.

If the SDK does not have an available endpoint to dispatch a scan creation request, it should retry until an endpoint becomes available.

Failures on Range Scan Create

Benign

- **Status::KeyEnoent (0x01)**
  - Description: The requested range is empty.
  - Not a real failure; the sub-stream should just return with 0 elements for this partition.

Retryable

- **Status::Ebusy (0x85)**
  - Description: Returned when the server refuses to start a new scan because too many scans are already in progress.
- **Status::Etmpfail (0x86)**
  - Description: When snapshot requirements are defined, this status indicates that the vbucket has yet to persist the required sequence number.

Retryable with special handling

- **Status::NotMyVbucket (0x07)**
  - Description: Additional to the usual cause of this status (e.g. vb is replica or not on this node)
  - Retry, but dispatch to the node hosting the active version of this vbucket.

Fatal

- **Status::Einternal (0x84)**
  - Description: The create can return this response for various runtime issues, logging should be generated by such issues.
  - Internal errors should always fail the stream.
  - #5 `InternalServerFailure`
- **Status::VbUuidNotEqual (0xa8)**
  - Description: If there is a vb-uuid mismatch it will result in this status.
  - The error is raised to the user and should fail the stream.
  - Happens if snapshot requirements are specified but the vbuuid of the requirements do not align with the server (i.e. hard failover)
  - #133 `MutationTokenOutdated`
- **Status::UnknownCollection (0x88)**
  - The collection was dropped.

    - If the collection is dropped mid-stream, fail the substream.
    - #11 `CollectionNotFound`
- **Status::NotStored (0x05)**
    - Note: since we are not using seqno_exists, this error is not returned. No need to expose it specifically.
    - Description: When snapshot requirements are defined, this status indicates that the sequence number which must exist, i.e. "seqno_exists":true, was not found.
    - This would be a non-recoverable error and needs to hard-fail the stream.

Any other status code is unexpected, and should be treated as a fatal error.

Failures on Range Scan Continue

Retryable with special handling

If these error conditions occur during a sampling scan, the substream should immediately complete successfully as if there were no more items available. This should not cause the overall stream to fail.

If these error conditions occur during a range scan, the substream must be replaced with a newly created subscan. The new scan should start at the last item received (exclusive), and end at the same point as the original end term. This restarting of the stream has the known effect of no-longer providing snapshot consistency of the scan; this is considered acceptable by the SDKs as the cross-vbucket nature of our scans already precludes providing this level of consistency.

- **Socket closed while streaming**
    - The error should not be propagated to the user-level; the user will just see increased latencies for the remaining items (up until timeout).

- **Status::NotMyVbucket (0x07)**
    - The requested vbucket no longer exists on this node. Happens during a rebalance.

Fatal (unless sampling scan)

These status codes indicate a serious problem that should result in the substream being terminated.
If they occur during a sampling scan, the failure should not be propagated to the user; instead, the substream should complete successfully as if no more items are available.

If these status codes occur during a range scan, they should be reported to the user and cause the overall stream to fail.

- **Status::KeyEnoent (0x01)**
    - No scan with the given uuid could be found.
    - Since the UUID just came from the create operation, this is an error condition and should fail the substream.
- **Status::Eaccess (0x24)**
    - The user no longer has the required privilege for range-scans.
    - If the privileges are revoked during a stream there is no real chance to get it going before the timeout hits - fail the substream.
    - #6 `AuthenticationFailure`
- **Status::UnknownCollection (0x88)**
    - The collection was dropped.
    - If the collection is dropped mid-stream, fail the substream.
    - #11 `CollectionNotFound`
- **Status::RangeScanCancelled (0xA5)**
    - The scan was canceled whilst returning data.
    - If the SDK canceled the scan because the user signaled they are no longer interested in the results, this status code is benign and should be ignored.
    - If the scan was canceled (via an unspecified request cancellation mechanism) while a different actor was still processing the results, then the cancellation error must be propagated to the user so they know the stream was canceled by another actor.
    - #2 `RequestCancelledException`

Fatal (always)

These error conditions are fatal regardless of scan type, and should cause the overall scan to fail.

- **Feature not available**
    - If a user attempts a scan but the server does not support the KV Range Scan feature, the SDK must report a `FeatureNotAvailable` error.
- **Status::Einval (0x04)**
    - The request failed an input validation check, details of which should be returned in the response. This indicates an SDK fault.
    - #3 `InvalidArgument`
- **Status::EBusy (0x85)**
    - The scan with the given uuid cannot be continued because it is already continuing.
    - The SDK should error the whole stream, as this can only be caused by an internal SDK fault.

Any other status code is unexpected, and should be treated as a fatal error.

Tracing Spans

Tracing spans for Range Scan are a bit more complicated than a typical request, since a whole flow of requests and responses across many partitions needs to be modeled.

The following enumeration describes the trace span hierarchy.

- (Optional) Parent Span passed in by the user
  - Overall "Range Scan" Operation Span (Identifier: `range_scan`)
    - One "Range Scan Partition N" Span per partition (Identifier: `range_scan_partition`)
      - One Range Scan Create Request (complete once response arrives) (Identifier: `range_scan_create`) per Create Request that is created.
      - N Range Scan Continue Requests (complete once all responses arrived) (Identifier: `range_scan_continue`) per Continue request that is created.

Span tags are as follows:

- `range_scan`
  - `num_partitions (number)`: number of partitions scanned
  - `scan_type (string)`: either "`range`" or "`sampling`"
  - if sampling:
    - `limit (number)`
    - `seed (number)`
  - if range:
    - `from_term (string, Base64-encoded term bytes)`
    - `to_term (string, Base64-encoded term bytes)`
  - `without_content (boolean)`
- range_scan_partition
  - `partition_id (number)`: the partition ID
- range_scan_create
  - `scan_type (string)`: either "`range`" or "`sampling`"
  - if sampling:
    - `limit (number)`
    - `seed (number)`
  - if range:
    - `from_term (string)`

            - `to_term (string)`
            - `from_exclusive (boolean)`
            - `to_exclusive (boolean)`
    - `without_content (boolean)`
- range_scan_continue
            - `range_scan_id (string)`
            - `item_limit (number)`
            - `byte_limit (number)`
            - `time_limit (number)`

Reference Material

- PRD
- KV-Engine Design

Open Questions / Todo

- ~~Batch time limit – Configurable? Default value? Interactions with timeout?~~
- [Brett] Should we specify how to cancel a Range Scan, or not expose RequestCancelledException to the user?

Language specifics

Go

- ScanResult does not return an error for expiry or cas, maintaining consistency with GetResult.

Revisions

- Revision #9 (2023-06-08; David Nault, Brett Lawson):
    - Removed `#131 RangeScanIdFailure` (throw CouchbaseException instead)
    - Removed `#132 RangeScanCancelled` (throw RequestCanceledException instead)
    - Removed `ScanTerm::minimum` and `ScanTerm::maximum`. Instead, the `from` and `to` fields of a `RangeScan` are optional; an absent `from` or `to` indicates the endpoint is unbounded.
    - Removed references to server-side implementation details that do not impact clients.

    - Removed "reading more than one document at a time" from the list of motivations, since a range scan is not expected to perform better than getting documents by ID using the existing KV `get` operation.
    - Noted that collation (using locale-specific rules for string comparison) is out of scope.
    - Noted that all requests related to the same stream must occur on the same KV connection.
    - Clarified ideal behavior for concurrent streams.
    - In languages that do not support exceptions, or if throwing an exception would not be idiomatic, `ScanResult` MAY return dummy values for `content`, `cas`, and `expiryTime` if `idsOnly()` returns true.
    - SDKs that apply the range scan timeout to each individual emission should omit `batchTimeLimit` from `ScanOptions`, and should always set the time limit to 90% of the client-side timeout. SDKs that handle timeouts differently should include `batchTimeLimit` in ScanOptions, with a default value of 0 (unlimited).
    - Added `concurrency` to ScanOptions. Defaults to 1. SDKs that have not implemented concurrent scans should omit this option from ScanOptions.
    - Removed threshold logging for range scan. Removed `kvScanThreshold` field from the threshold logging config.
    - Clarified that if the SDK does not have an available endpoint to dispatch a scan creation request, it should retry until an endpoint becomes available.
    - Clarified that if multiple concurrent substreams fail permanently for different reasons, the SDK arbitrarily selects one of the failures to report to the user as the cause of the overall stream failure.
    - Categorized each error condition as "Benign", "Retryable", "Retryable with special handling", "Fatal (unless sampling scan)", or "Fatal (always)".
- Revision #8 (2023-05-16; David Nault):
    - Make PrefixScan a first-class part of the API.
- Revision #7 (2023-04-20; David Nault):
    - Changed ScanTerm term from byte[] to String. Replaced 0xff with the maximum Unicode code point U+10FFFF, represented in UTF-8 as the 4-byte sequence: 0xf48fbfbf.
    - To expedite the release of server resources, the client cancels any scan it does not complete.
    - Removed sorting and associated options.
    - To avoid overwhelming the server, a client should stream from no more than one of a node’s vbuckets at a time (per scan). The simplest way to do this is to stream from only one vbucket at a time (per scan).
- Revision #6 (2022-11-29; David Nault):
    - Relaxed the timeout specification to reflect the lowest common denominator behavior across SDKs: Must raise UnambiguousTimeout error if first item is not received before timeout expires, or some variation on that theme.
    - Clarified that batchByteLimit and batchItemLimit are uint32.

    - Implementations that wish to expose the batch time limit parameter should use the name `batchTimeLimit`.
    - During a sampling scan, if a partition scan encounters an error, the partition scan reports completion instead of raising an error or retrying.
- Revision #5 (2022-11-28; David Nault):
    - ScanTerm.exclusive defaults to false.
    - ScanTerm.term should be stored as a byte array, not a string. Users must be able to pass either a byte array, or a string which is converted to a byte array using UTF-8.
    - RangeScan.from defaults to ScanTerm::minimum()
    - RangeScan.to defaults to ScanTerm::maximum()
    - Implementations should provide a helper method to create a `RangeScan` for a document ID prefix. Pseudocode: `(from = prefix inclusive, to = prefix.toByteArray() + 0xff exclusive)`.
    - SamplingScan.seed defaults to a random number.
    - ScanOptions.withoutContent renamed to idsOnly.
    - ScanResult has a new accessor, `idOnly`; returns the value of `idsOnly` from the scan associated with the result.
    - SDK reports `FeatureNotAvailable` error if the server does not support KV range scan.
    - Clarified that the timeout applies to time between start of scan and receipt of first item. An implementation may also apply the timeout to the interval between subsequent emissions.
- Revision #4 (2022-10-20; Michael N.):
    - Set default batchByteLimit to 15000 (15k) and the default batchItemLimit to 50 based on discussion with kv engine team.
- Revision #3 (2022-10-19; Michael N.):
    - Added `batchByteLimit(int)` and `batchItemLimit(int)` to the `ScanOptions`.
- Revision #2 (2022-10-06; Michael N.):
    - Clarified `MutationTokenOutdated` failure after new status code introduced on the server side (0xa8)
- Revision #1 (2022-09-26; Michael N.):
    - Completed initial draft

Signoff

Language	Team Member	Signoff Date	Revision
.NET	Jeffry Morris		

Language	Team Member	Signoff Date	Revision
C++	Sergey Avseyev		
Go	Charles Dixon		
Java/Kotlin	David Nault	2023-06-08	9
Node.js	Brett Lawson		
PHP	Sergey Avseyev		
Python	Jared Casey		
Ruby	Sergey Avseyev		
Scala	Graham Pople	2022-12-01	6
Spring Data	Michael Reiche		