

NoSQL ドキュメント指向 データベース Couchbase Server ファーストステップガ イド

河野泰幸 著

2021-10-01 版 発行

まえがき

本書の構成

本書は、表現や内容、あるいは直接的に想定されている読者層について、傾向の異なる幾つかの部分からなります。

第1章は、「プロローグ」と題されており、この章だけが、二人の登場人物の対話の形式で書かれています（底本には含まれていない新規書き下ろし部分になります）。この章では、実践的な観点から特に重要と思われる情報を効率的に、そして頗るくば親しみ易さと共に、伝えることが目指されています。

プロローグに続き、ここからが本編という意図により、第2章は「はじめに」と題されています。ここでは、以降の章を読み進めるにあたっての前提を整理しています。

第3章もまた、第4章以降とは若干趣が違う内容となっています。ここでは、Couchbase Server の存在意義を伝えることに主眼が置かれています。そのため、Couchbase Server がその潮流の一部であるような、現在の技術的背景の記述についても多くの充てています。Couchbase Server の存在意義を一面的に語ることができないことを反映し、この章を構成する節は相互に独立し、基本的には、別々に読まれても成立する内容になっています。

第4章以降については、一般的な解説書の構成と大きく異なるものとなっているかと思います。

第4章では、初めて Couchbase Server を利用するユーザー向けに、検証・

開発環境を構築する方法を説明しています。

第5章から第7章は、データベース利用者（開発者）にとって必要となる情報、すなわちデータの論理構造、クエリ操作、そしてクエリ実行の前提となるインデックスに関する章となっています。

第8章から第10章は、アーキテクチャー、Couchbase Server を構成する要素（コンポーネント）、そしてセキュリティについて解説しており、Couchbase Server の全体像把握に資する内容です。

第11章から第15章は、フェイルオーバーやバックアップ等の運用や環境設定に関する内容に充てられています。

第16章は、Couchbase Server SDK と、外部システム連携のためのコネクターを紹介する短い章となっています。

最後の章となる17章は、具体的なアプリケーション開発を主題としています。

リポジトリについて

本書で紹介しているサンプルアプリケーションのソースコードを以下の URL で公開しています。また、このリポジトリを、本書に関する情報共有のために、必要に応じ更新・活用する予定です。

https://github.com/YoshiyukiKono/Couchbase_Server_First_Step_Guide.git

表記関係について

本書に記載されている会社名、製品名などは、一般に各社の登録商標または商標、商品名です。会社名、製品名については、本文中では®、™マークなどは表示していません。

免責事項

本書の文責は著者にあり、所属する組織とは関係ありません。

また、本書に記載された内容は、情報の提供のみを目的としています。正確かつ適切な情報を届けるため配慮を尽くしていますが、本書の内容の正確性、有用性等に関しては、一切保証するものではありません。したがって、本書の情報を用いた開発、運用、コンサルティング等、いかなる実践も必ずご自身の責任と判断によって行なってください。本書の情報を参照した行為の結果について、著者はいかなる責任も負いません。

底本について

本書籍は、2021年7月10日から25日まで開催された技術系同人誌即売会「技術書展11」で頒布された「オープンソースJSONデータベースCouchbase Server ファーストステップガイド」を底本とし、2021年7月末にリリースされたCouchbase Server 7.0にあわせた情報の更新を行った他、全面的に改訂しています。

目次

まえがき	iii
本書の構成	iii
リポジトリについて	iv
表記関係について	iv
免責事項	v
底本について	v
第1章 プロローグ: データベース選択を巡る対話	1
第2章 はじめに	11
2.1 Couchbaseについて	11
Couchbase Serverについて	12
エディションとオープンソースプロジェクト	13
2.2 関連データベース技術	14
ドキュメント指向データベース	14
キーバリューデータベース	14
インメモリデータベース	15
リレーショナルデータベース	15
クエリとインデックス	15
トランザクション	16

目次

第3章 Couchbase Server とは何か	17
3.1 Couchbase Server の歴史	17
インメモリ KVS から JSON データベースへ	18
クエリ	19
全文検索	19
イベント処理	20
アナリティクス	20
スコープとコレクション	21
運用やセキュリティ機能の拡充	22
3.2 NoSQL と現在の技術動向	22
DX (Digital Transformation)	23
SoE (System of Engagement)	23
NoSQL の登場まで	24
SoE と NoSQL	25
SoE のためのデータベースとしての Couchbase Server	28
3.3 メモリーファーストアーキテクチャー	29
インメモリ DB と永続化	29
データ永続化パターン	30
データ復旧アプローチ	31
Oracle Coherence と Redis の永続化手法	31
永続化機構を備えたインメモリ DB としての Couchbase Server	32
3.4 JSON データベース	34
JSON の特徴	34
JSON と Web フロントエンド技術との親和性	35
Web アプリケーション開発と高い親和性を持つ Couchbase Server	36
3.5 トランザクションと NewSQL	36

目次

	ドキュメント指向データベースにおけるトランザクション	
	の原子性	36
	マルチドキュメントトランザクションの必要性	37
	Couchbase Server の分散トランザクション	38
	NewSQL	39
	楽観的ロックの何が問題か	41
	性能を犠牲にしないトランザクション適用の柔軟性を持つ	
	Couchbase Server	42
3.6	第一正規化制約を超えて	43
	NoSQL のカテゴリーについて	43
	「準(半)構造化データ」という呼称について	44
	Couchbase Server と RDB のデータモデルの違いと共通点	44
	RDB のスーパーセットとしての Couchbase Server	45
	第 4 章 Couchbase Server を使ってみる	47
4.1	インストール	47
	ダウンロード	47
	インストール手順	47
4.2	Docker コンテナ利用	48
	初回実行	48
	ログの確認	48
	イメージの確認	49
	プロセスの確認	49
	コンテナの停止と再起動	49
	コンテナ上のコマンド実行	50
	コンテナとイメージの削除	51
4.3	クラスター初回構築	51
	Web コンソールへのアクセス	51
	クラスターセットアップウィザード	52

目次

	クラスター設定	55
4.4	Web コンソールの使い方	57
	バケット作成	57
	スコープ作成	60
	コレクション作成	61
	ユーザー作成	62
	ファイルからのデータ登録	64
	サンプルデータ利用	65
	クエリ実行	66
	インデックス作成・確認	67
4.5	クラスター構成変更	68
	ノードの追加	68
	ノードの削除	70
第5章	データ	71
5.1	ドキュメント	71
	概要	71
	キー	72
	値	73
	メタデータ	73
5.2	バケット	73
	概要	73
	種類	74
5.3	スコープとコレクション	75
	概要	75
	コレクションの存在意義	75
	スコープの存在意義	76
	機能	77
	基本仕様	78

目次

作成・管理	78
デフォルトのスコープとコレクション	79
ユーザー定義のスコープとコレクション	79
5.4 有効期限設定	80
バケットおよびコレクションレベルの設定	80
デフォルト設定	81
第6章 N1QL クエリ	83
6.1 基本構造	83
コンストラクションオペレーター	83
ネスティッドオペレーター	83
エスケープ文字	84
クエリコンテキスト	84
6.2 ドキュメントキーの利用	86
USE KEYS	86
META().id	87
6.3 フィールドから値のみを取り出す	89
RAW ELEMENT VALUE	89
6.4 複数の値をまとめて扱う	91
IN	91
WITHIN	92
6.5 配列へのアクセス	93
ARRAY	94
FIRST	94
WHEN	94
6.6 配列を検索条件に利用	96
ANY	96
EVERY	96
6.7 サブドキュメントの階層操作	97

目次

NEST	97
UNNEST	99
6.8 データ型	101
MISSING	102
NULL	103
6.9 比較演算	103
IS NULL	103
IS MISSING	104
IS VALUED	104
6.10 結合 (JOIN)	105
ANSI JOIN	106
ルックアップ JOIN	107
インデックス JOIN	107
6.11 ユーザー定義関数	109
ユーザー定義関数の種類	109
関数呼び出しの種類	110

第7章 インデックス	113
7.1 概要	113
用語法に関する注釈	113
来歴	114
RDB におけるインデックスとの違い	115
ストレージモデル	115
インデックス情報の確認	116
7.2 プライマリインデックス	117
概要	118
定義方法	118
7.3 セカンダリインデックス	120
概要	120

目次

定義方法	121
ドキュメントキーの利用	122
7.4 配列インデックス	122
要素オブジェクトの特定フィールド指定	123
要素オブジェクトの複数フィールド指定	125
特定条件で選択された配列要素を利用	125
要素オブジェクトの全てのフィールドを利用	126
7.5 インデックス論理設計	127
複合 (Composit) インデックス	127
カバリング (Covering) インデックス	128
部分 (Partial) インデックス	128
関数 (Functional) インデックス	129
適応 (Adaptive) インデックス	130
7.6 インデックス物理設計	131
重複 (Duplicate) インデックス	131
パーティション	133
7.7 先行属性マッチと性能最適化	134
カーディナリティ	134
Data サービス API との併用	135
第 8 章 アーキテクチャ	137
8.1 分散アーキテクチャー	137
ピアツーピアトポロジー	137
ピアツーピアアプリケーション	138
マルチディメンションナルスケーリング (MDS)	141
オンラインリバランス	143
データベースチェンジプロトコル (DCP)	143
vBucket	144
8.2 スマートクライアント	144

目次

8.3	クラスターマップ	144
8.3	ブートストラップ	147
8.3	XDCR(Xross Data Center Replication)	148
8.3.1	XDCR は、なぜ必要か	148
8.3.2	XDCR は、どのように動作するか	148
8.3.3	チェックポイント	149
8.3.4	レプリケーション指定とフィルタリング	150
8.3.5	レプリケーション優先度	150
8.3.6	ネットワーク帯域幅節約のための最適化	151
8.3.7	運用上の注意	151
8.4	サーバーグループ	151
第 9 章	Couchbase Server の構成要素	153
9.1	Data サービス	153
9.1.1	ディスパッチャー	153
9.1.2	KV エンジン	154
9.1.3	スケジューラー	155
9.2	Query サービス	155
9.2.1	リスナー	155
9.2.2	クエリプロセッサ	156
9.2.3	データストア	156
9.2.4	他のサービスとの連携	156
9.3	Index サービス	157
9.3.1	スーパーバイザー	157
9.3.2	プロジェクトとルーター	158
9.4	クラスターマネージャー	159
9.4.1	ns-server	160
9.4.2	babysitter	160
9.5	マスターサービス	161

目次

クラスタートポロジー管理	161
ノード障害検出	162
バケット管理	162
管理、統計、およびロギング	163
第 10 章 セキュリティ	165
10.1 主要なセキュリティ要素	165
認証 (Authentication)	165
認可 (Authorization)	165
監査 (Auditing)	166
暗号化 (Encryption)	167
10.2 セキュリティ管理項目	167
ネットワークアクセス	167
証明書	167
ログリダクション	167
セッションタイムアウト	168
ポートマッピング	168
10.3 認証	169
ビルトイン管理者アカウント	169
パスワードポリシー設定	170
外部アクセシビリティ管理	170
Web コンソールでの認証	171
アプリケーションの認証	171
証明書ベースの認証	172
10.4 認可	173
データベースユーザーの種類	173
ビルトインロール	173
10.5 暗号化	174
通信データ	174

目次

永続化データ	175
システム機密情報	175
JSON ドキュメントフィールド	176
第 11 章 フェイルオーバー	177
11.1 フェイルオーバー概観	177
計画的なノード削除（グレースフルフェイルオーバー）	177
障害時のノード削除（ハードフェイルオーバー）	178
フェイルオーバー実行方法と考慮点	178
自動フェイルオーバーにおける問題	179
自動フェイルオーバーにおける制約	179
保守運用	180
外部システム連携	180
11.2 自動フェイルオーバー	181
有効化設定	181
実行条件	181
待機時間	182
通知	182
サービス固有ポリシー	183
グループフェイルオーバー	183
ディスク障害に対するフェイルオーバー	184
11.3 手動フェイルオーバー	184
Web コンソール	185
コマンドラインツール	185
REST API	185
第 12 章 運用	189
12.1 監視	189
Web コンソール	189

目次

REST API	190
コマンドラインツール	190
アラート E メール	191
Prometheus 連携	191
12.2 バックアップ	192
概要	192
Backup サービス	193
コマンドラインツール	193
第 13 章 コマンドライン操作	195
13.1 基本	195
インストール場所	195
13.2 クラスター管理: couchbase-cli	196
基本構文	196
サブコマンド	196
13.3 クエリ: cbq	199
実行例	199
13.4 データのインポート: cbimport	200
概要	200
基本構文	200
ホストフォーマット	201
JSON データセットフォーマット	202
CSV データセットフォーマット	203
ドキュメントキー	203
インポート先の指定	204
データ型推論	204
利用例	204
13.5 データのエクスポート: cbexport	205
概要	205

目次

基本構文	205
データセットフォーマット	206
13.6 バックアップ: cbbackupmgr	206
バックアップリポジトリの作成	206
バックアップ構成の確認	208
バックアップの作成	208
増分バックアップ	211
バックアップの復元	213
バックアップのマージ	214
バックアップリポジトリの削除	216
第 14 章 ユーティリティ	219
14.1 概要	219
インストール	219
共通オプション	220
クライアントオプション	221
14.2 ストレステストツール	222
概要	222
pillowfight	222
n1qlback	225
14.3 その他のツール	226
第 15 章 環境設定	227
15.1 メモリからのデータ排出 (Ejection)	227
概要	227
排出方法オプション	229
選択基準	229
設定変更時の注意点	230
メモリ消費状況の把握	230

目次

15.2	コンパクション	232
	概要	232
	運用上の留意点	233
	オートコンパクション	233
15.3	代替アドレス	234
	概要	234
	Couchbase Server の特殊性との関係	234
	外部ネットワークからの接続	235
	デュアルネットワーク	237
第 16 章 SDK と外部システム連携		241
16.1	SDK	241
	サポートされるプログラミング言語	241
	拡張ライブラリ	242
16.2	外部システム連携	242
	コネクター	242
第 17 章 Node.js アプリケーション開発		245
17.1	なぜ、Node.js と Couchbase Server の組み合わせなのか .	245
	JSON データの利点	245
	クエリ言語の利点	246
17.2	キーバリュー操作	246
	CRUD 操作	247
	サブドキュメント操作	248
17.3	データ一貫性 (Consistency)	253
	分散アーキテクチャーにおけるインデックス	253
	インデックス更新から見た RDB との違い	253
	キーバリュー操作とクエリ	254
	インデックス更新が非同期であることの影響範囲	254

目次

Read Your Own Writes (自分自身が書いたデータを読む)	255
永続化や複製の場合と比べたインデックスの特殊性	256
一貫性を保証するためのオプション	256
17.4 サンプルアプリケーション紹介	259
CEAN スタック	259
アプリケーション概要	259
アプリケーション実行事前準備	260
アプリケーション利用方法	261
プログラム解説	262
17.5 Node.js SDK を開発に利用する	265
動作確認環境	265
プロジェクト準備手順	266
17.6 ODM フレームワーク Ottoman.js	267
Ottoman.js とは	267
機能概要	267
17.7 開発の実践に向けて	268
コミュニティ	269
Couchbase Labs とサンプルアプリケーション	269
無償オンライントレーニング	270
ドキュメント、API Docs、ソースコード	270
付録 A NoSQL 性能評価: MongoDB、Cassandra との比較	271
A.1 ベンチマーク概要	272
検証パターン	272
クラスター構成	273
サーバーハードウェアスペック	273
共通条件	273
データベース固有の構成	273
A.2 検証パターン別詳細	274

目次

ワークロード A: 更新処理	275
ワークロード E: レンジスキャン	276
クエリ 1: ページネーション (OFFSET と LIMIT による フィルター)	279
クエリ 2: JOIN (テーブル結合)	280
A.3 結論	283
付録 B データマイグレーションツール	287
B.1 GlueSync	287
B.2 Couchgres	288
B.3 SqlServerToCouchbase	288

第1章

プロローグ：データベース選択を巡る対話

ここはある技術系ミートアップ会場、有志による LT(ライトニングトーク)が行われた後のフリータイムに、各所で思い思いの会話がなされています。

その一角で、また新しい対話が始まります。

「先ほど発表された内容、とても勉強になりました」

「ありがとうございます。あなたの新しいアプリケーションの構想も、なかなか興味深かったですよ」

「そうですか！ まだ、アイディアでしかないんですけどね。ところで、少し相談させてもらってもいいですか？」

「自分でよければ」

「ありがとうございます！ かなり漠然とした相談になってしまいますが、先ほど発表したアプリケーションのプロトタイプを開発するに当たって、どのデータベースを使うか悩んでいます。データベースについて、広い知識をお持ちのようでしたので…」

「広いかどうかは分からぬけど。どんな部分で悩んでいますか？ 右も左

第1章 プロローグ：データベース選択を巡る対話

も分からぬという感じではなさそうだけど」

「あまり深く考えずに、とりあえず作り始めるとしたら、何か適当な RDB を使うと思うんですよね...」

「はい」

「プロトタイプを作っている間、データモデルについては試行錯誤すると思うんです。それだけなら、テーブル定義を修正しながら進めてもいいんですが...」

「ふんふん」

「プロトタイプとは言っても、可読性と保守性の高いコードを維持したいんです。これは絶対に！」

「いいですね」

「だから、ドメインエンティティを素直に反映したデータモデルでコーディングを進めたくて、RDB でデータを扱う際の制約とは切り離しておきたいんです」

「分かります」

「それで、ORM(オブジェクトリレーションナルマッピング)について考え始めると、特定の技術への偏りと、そのための負担が、煩わしくなってくるんですよね」

「そういうことですね。今の話を聞くと、すでに何か RDB 以外で検討しているようだけど、どうですか？」

「はい。ドキュメント指向データベースというんですか？ JSON を保存できるデータベースなら、アプリケーションで扱っているエンティティと同じ構造のデータを保存できると思ったんですが...」

「そうですよね。だけど？」

「それはそれで、実際に開発を始めようとすると、また別の技術的な偏向というか、負担が気になるんですよね...」

「ドキュメント指向データベースは、開発の面について、SQL のように標準化されているわけではないですからね」

「そうなんです！ 仕方がないんですかね？」

第1章 プロローグ：データベース選択を巡る対話

「それでは、まず確認ですが、クラウド利用については、どう考えていますか？」

「もちろん、いずれクラウドは有効に活用したいと思っていますが、今は、特定のクラウドベンダーに縛られたくはなくて。開発中は、ラップトップで完結させたいと思っています」

「それでは、クラウドベンダー独自の技術は自ずと除外されるとして…」

「何か思い当たりますか？」

「Couchbaseって、聞いたことあるかな？」

「う～ん、名前だけは聞いたことがあるような…Apacheプロジェクトでしたっけ？」

「いや、Apache CouchDBというのははあるけど、それとは別のオープンソースのデータベースです」

「知ったかぶりして恥ずかしい！」

「いやいや。多分、名前はどこかで聞いていたんでしょうね」

「あらためて、何も知らないものとして、教えてもらえますか？」

「それでは、お話を伺って、興味を持たれるかもしれないと思った Couchbase の特徴について挙げていきますね」

「お願ひします！」

「まずは、JSON ドキュメントを扱うドキュメント指向データベースの中でも、多くのプラットフォームで稼働するオープンソースのテクノロジーであるということ」

「それは、私の希望しているところでもあります」

「それから、開発の面で、SQL を使えるところ…」

「そこを詳しく教えてもらえますか！ いろいろな RDB 以外のデータベースがクエリを使えると謳っていて、少し調べると、なんだか…」

「期待していたものと違う」

「そうなんです。よく理解できていないだけなのかもしれません…逆に、RDB が JSON に対応した、というケースもありますよね。どちらも、なんだか、しっくりこないんです」

第1章 プロローグ：データベース選択を巡る対話

「SQL を JSON 用に拡張する時の 1 つの方法として、JSON を扱うための関数を追加する、というものがあります。この場合、純粋な SQL のシンタックスは保持されますが、JSON データから発想している開発者にとっては、期待しているものとのギャップがあるのではないかと思います」

「はい」

「一方で、関数の追加ではなく、SQL のシンタックス自体を JSON データ構造用に拡張するというアプローチがあります」

「全然違いますね」

「JSON の特徴として、配列やネストされたデータが扱えるということがありますよね」

「はい、分かります」

「例えば、SQL のクエリで直接、配列の表現が使えたり」

「ははあ」

「オブジェクトのプロパティにドット表記でアクセスできたり、普段から開発者がプログラミングで用いている表現が使えれば直感的ですよね」

「そうですね」

「先ほど、期待していたものと違う、という話がありましたが、何が一番、ギャップとして大きいと感じますか？」

「当然の質問だと思うんですが、正直、よく説明できないんですよね...」

「意地悪な質問だったかもしれませんね。今、話したようなデータ構造の違いだけなら、関数でも吸収できるでしょう。だけど、データ単体ではなく、複数のデータを扱うことを考え始めると、話が違ってきます。端的にいえば、データを結合できるかどうかが問題になると思うんです」

「何となく分かるような気がします」

「JSON が、ネストした構造を持てるからと言って、関係する全てのエンティティを 1 つのドキュメントとして保存するのは現実的ではありませんよね」

「確かに。できることと、やるべきことの違いですね」

「参照キーでデータ間の関係を表現するのは、リレーションナルモデルの中

第1章 プロローグ：データベース選択を巡る対話

心にある発想ですが、データモデルの設計としては、RDB のテーブルに限らず、他のデータ構造でも適用可能な考え方です」

「それはそうですね」

「後は、複数のデータを結合してデータを取り出せるかどうか？」という話になってきます」

「ふむふむ」

「その点、Couchbase の N1QL(ニッケル) は、期待されるものを実現しています」

「ニッケル、ですか？」

「Couchbase のクエリの呼称です。Non 1st normal form Query Language の略で、N1QL と書いて、ニッケルと読みます」

「なるほど。JSON は、RDB のように第一正規化が必要ないから、非第一正規形クエリ言語、というわけですね」

「そうそう。今、第一正規化が必要ない、と言われましたよね。必要ないだけで、逆に第一正規化されている、つまり普通の表形式のデータであっても何の問題もない、というのがポイントです」

「というと？」

「例えば、既存のリレーションナルデータベースで扱っているデータを、データモデルを変えずに、JSON データとして Couchbase に移植した場合には、RDB で使っていたのと全く同じ SQL を使う、といったことも可能になってきます」

「へえ。なんとなく、技術的に偏った方向に振れすぎる、という心配を解消してくれそうな気がしてきました」

「うん、極端な話、Couchbase でプロトタイプを作った後、RDB に移行するということも、設計次第で成立させられると思いますね。メリットがあるかどうかは別として」

「面白いですね。実際にはありそうではなくても、そういうことも考えられるくらい汎用性があって、標準的な知識を活かすことができる、というのは大事だと思います」

第1章 プロローグ：データベース選択を巡る対話

「標準という意味では、SQL++^{*1}という準構造化データへのクエリに関する標準規格があって、Apache プロジェクトにも AsterixDB^{*2}という SQL++ を採用しているデータベースがあります。Couchbase の N1QL は、SQL++ と全く同じものではないけど、多くの共通する部分を持っています^{*3}」

「それは、悪くない情報ですね」

「ご相談に直接関係する部分としては、こんなところですかね。お役に立てたでしょうか？」

「はい、ありがとうございます」

「あと、Couchbase を使った Node.js のサンプルアプリケーションを持っていますが、見てみますか？」

「ぜひ！」

「Couchbase とアプリケーションを起動しますね… 画面はこんな感じです。ご覧の通り、とても単純なユーザー管理アプリケーションです」

「これくらい簡潔だと、内部で行われていることを理解しやすそうですね」

「これがソースです」

「ユーザーのリスト表示に使っているクエリがこれですね。確かに普通の SQL と同じですね」

「他には、ユーザーの追加と削除しか機能がありませんが、それはこの辺りです」

「INSERT や DELETE のクエリを使うわけではないんですね」

「使うこともできるけどね。データを直接操作して済むなら、わざわざ文字列でクエリを渡して、サーバーでそれを解析してから実行して… そんな必要はないということですね」

「ORM を使っている時と似ているかも」

「こういう部分、ORM では SQL を隠しますよね。ここではそういった

^{*1} <https://arxiv.org/abs/1405.3631>

^{*2} <https://asterixdb.apache.org/>

^{*3} 上に紹介した論文には、「syntactic sugar over SQL++」という表現が見られます。

第1章 プロローグ：データベース選択を巡る対話

ラッパーを使っていないので、リソース消費の上でも、知識習得の面でも、余計なオーバヘッドを避けることができる、と言えるんじゃないかな」

「なるほど」

「反対に、データを検索する時には、SQLの方が開発者にとって直感的なのに、ORMを使うために、余計な苦労を強いられることがあるんじゃないですか？」

「分かる気がします。私がRDBとORMの組み合わせを前提に開発を進めることに二の足を踏んでいたのも、そんなところです」

「ここまでとのところ、どうですか？」

「プログラムを見せていただいたので、具体的にイメージが掴むことができました。コード量もこれだけで、他にフレームワークやライブラリは使っていないんですよね？」

「はい。データベースアクセスに関して、Couchbaseライブラリのみを使っています。はじめにクラスターへの接続を確立しておいて、必要なところで、データ表示のためのN1QLクエリと、データ操作のためのAPIコールを行っています」

「ここで使っているのは、とても基本的なクエリですが、自分でもっと複雑なことをする場合も、SQLが使えるから応用が効きそうです」

「役に立ちそうですか？」

「はい！何より、シンプルなのが気に入りました」

「それはよかった。そうそう、今見てもらったように、開発環境をラップトップで完結させたい、という要望にも適していると思います」

「ちなみに、本格的な運用では、どうなりますか？」

「NoSQLという言葉は聞かれたことがあると思いますが、Couchbaseは、NoSQLにカテゴリ化されるデータベースです。他のNoSQLデータベースと同じように、複数のノードからなるクラスターとして構成されます」

「ああ、それ！そのクラスターというところなんですが…やっぱり構築とか大変ですよね？」

「そんなことありませんよ。Couchbaseの良いところは、他の分散データ

第1章 プロローグ：データベース選択を巡る対話

ベースのように、マスターとワーカーとか、レプリカセットのような、特別な構成が必要ない事です。全てのサーバーに同じものをインストールします」

「すみません、構築が他の NoSQL より簡単ということですが、今一つ、どこが違うかが分からなくて...」

「データベースを複数台で構成する利点として、データを複製して持つことと、データを分散して持つことがあります。この違いについては、分かりますか？」

「データが複製されているから、1つが失われても大丈夫だし、データが分散されているから、処理も分散できる、ということですよね」

「そうですね。そして、多くの NoSQL では、データを分散する方法が、データのキーの設計に関係してきます」

「キーをどのように決めるかによって、複数のサーバーへのデータの散らばり方が変わってくるということですね。NoSQL では、どんな検索条件が使われることになるかを踏まえてデータベースを設計する必要がある、という話を聞いた覚えがあります」

「Couchbase では、物理的なデータの分散方法とキーの設計が独立しているのが特徴です。つまり、キーの設計による性能への影響がない、ということですね。これは、データの物理配置に影響を受けないアーキテクチャーになっているからですが、それだけでなく、他のデータベースと比べて、性能面で優位なアーキテクチャーで... という話は、長い話になってしまふので置いておくとして... データ設計を行うアプリケーション開発者にとっては、分散環境の構成に関する知識を前提とする必要がない、という利点があります」

「アーキテクチャーについての話も興味はありますが、まずはプロトタイプ開発なので、またそのうちに... データの論理設計と物理設計が切り離されているのはいいですね！」

「それで、どこが違うかが分からぬという話に戻ると、確かに本番環境の構築となると神経を使うところはあると思いますが、Couchbase の場合、

第1章 プロローグ：データベース選択を巡る対話

開発用に、1ノードで構成されたクラスターを使っている時と、本番環境で、複数ノードのクラスターを構築した時とで、論理的な構成が変わらないことが、開発される立場から見られてもそうであるように、構築する立場の人にとっても、お言葉を借りるなら、シンプルさを提供してくれます」

「なるほど、そういうことですね」

「他に気になるところ、ありますか？」

「今は大丈夫です。簡単に使い始められそうなので、まずは触ってみたいと思います」

「そうですか。それでは、私からもいいですか？」

「何でしょう？」

「LTで発表されていたアプリケーションについて、聞いてみたいことがあります...」

「何でも聞いてください！」

それぞれの関心を交差させながら、会話はさらに続くようです...

第2章

はじめに

2.1 Couchbaseについて

Couchbaseという名称で名指される技術には、大きくいって、次の2つがあります。

- Couchbase Server
- Couchbase Mobile

Couchbase Serverは、分散JSONドキュメントデータベース(distributed JSON Document database)です(ここでの英語による引用は、Open Source Projects from Couchbase^{*1}より)。Couchbase Mobileは、Couchbase LiteとSync Gatewayという2つの技術を包含する呼称です。Couchbase Liteは、モバイル端末および組み込みデバイス用のネイティブドキュメントデータベース(native document database for mobile and embedded devices)です。Sync Gatewayは、Couchbase MobileのデータをCouchbase Serverと同期するために使うことができるコンポーネント(components to sync to Couchbase Server)です。

^{*1} <https://developer.couchbase.com/open-source-projects/>

■コラム: Couchbase Server と Couchbase Lite

Couchbase Lite は、組み込み用ネイティブデータベースとして、プラットフォーム毎に開発されています。そこから明らかな通り、Couchbase Server と Couchbase Lite とは、それぞれ異なる技術を基盤としており、「ドキュメント指向データベース」であることは共通しているものの、Couchbase Server に関する本書の記述を、Couchbase Lite へ援用することはできません。

Couchbase Serverについて

Couchbase Server は、ドキュメント指向データベースにカテゴライズされる NoSQL データベースであり、シェアードナッシング型の分散アーキテクチャーを持ちます。

NoSQL と呼ばれるデータベースは、ビッグデータの集計処理や、IoT のようなリアルタイムかつ大量のデータ投入等、データベース毎にカバーするユースケースが異なっていますが、Couchbase Server はデータに対して、リード・ライトの両方のアクセスを、低遅延で実現する必要のある（つまり、インタラクティブな）アプリケーションのバックエンドとして用いられるために設計（最適化）されています。

■コラム: 本書で扱う Couchbase Server のバージョン

本書の記述は、2021 年 7 月にリリースされた、Couchbase Server 7.0^{*2}に基づきます。なお、本書執筆時点の最新バージョンは、2021 年 9 月にリリースされた、バグフィックスを含むメンテナンスリリースである、Couchbase Server 7.0.1^{*3}です。

エディションとオープンソースプロジェクト

Couchbase Server は、エンタープライズエディションとコミュニティエディションの、2つの形態でバイナリが提供されており、その基盤としてオープンソースプロジェクト^{*4}が存在しています。

コミュニティエディションは、一部の特別な機能を除いて、エンタープライズエディションの主要な機能を利用することができます。それらの機能について移植性が確保されています。つまり、コミュニティエディションを利用しているアプリケーションは、コードを変更することなく、エンタープライズエディションに移行可能です。一方で、エンタープライズエディションには、エンタープライズレベルの高い性能要件を満たすために、内部機構においてコミュニティエディションとはいくつかの違いが存在しています。また、コミュニティエディションでは、クラスターを構成するノード数についても制限（最大5ノードまで）があります。

コミュニティエディションは、エンタープライズエディションのメジャーリリースと同時にリリースされますが、マイナーバージョンは、数か月遅れてリリースされ、メンテナンスリリースについても、限定されています。

■コラム: エディションに関する本書の記載方針

本書の記述は、基本的にコミュニティエディションに基づいています
が、エンタープライズエディション固有の機能についても、エディションによる差異を示しながら紹介しています。

^{*1} <https://docs.couchbase.com/server/current/release-notes/relnotes.html#release-700>

^{*2} <https://docs.couchbase.com/server/current/release-notes/relnotes.html#release-701>

^{*4} <https://github.com/couchbase>

2.2 関連データベース技術

本書の記述は、データベース一般、特にリレーションナルデータベースについての知識を前提としています。

また、Couchbase Serverとの比較対照のために、他のNoSQLデータベースについて触れる場合、個々のデータベースについて精通していることは想定していませんが、NoSQL全般に関する一定の知識を前提としている部分があります。

ここでは、このような関連するデータベース技術について素描を試みつつ、詳細情報への参照先を掲載します。

ドキュメント指向データベース

ドキュメント指向データベース (Document-oriented database^{*5}) は、ドキュメントストアとも呼ばれます。ドキュメント指向データベースというカテゴリーには、XMLなどの様々なフォーマットを扱うデータベースが含まれますが、Couchbase Serverは、JSON形式のデータを扱うドキュメント指向データベースです。

キーバリューデータベース

キーバリューデータベース (Key-value database^{*6}) は、キーバリューストアとも呼ばれ、KVSという略称も広く用いられています。連想配列、ディクショナリ、あるいはハッシュテーブルとも呼ばれるデータ構造で、データを管理するデータベースです。

^{*5} https://en.wikipedia.org/wiki/Document-oriented_database

^{*6} https://en.wikipedia.org/wiki/Key%E2%80%93value_database

インメモリデータベース

インメモリデータベース (In-memory database^{*7}) は、メモリ内でデータを管理することによって、高速な応答性能を実現するデータベースです。一般に、データベースが、データ永続化装置 (ハードディスク等) を用いた永続性を保証しているのに対して、インメモリデータベースを利用する際には、永続化の要件は必要に応じ外部で実現されます。

リレーションナルデータベース

リレーションナルデータベース (Relational database^{*8}) は、関係データベースとも呼ばれ、関係モデル（リレーションナルデータモデル）に基づいて、データを扱います。

リレーションナルデータベースでは、データは（複数の）表として管理され、この表は「テーブル」と呼ばれます。一般に、表は行（カラム）と列（ロウ）からなりますが、テーブルの定義により決定される、単位データ（レコード）を構成する要素（フィールド）は「カラム」と呼ばれ、レコードは「ロウ」と呼ばれます。

クエリとインデックス

データベースにおけるクエリ (Query^{*9}) は、データベース管理システムへの問合せ一般の表現として普通名詞化されています。リレーションナルデータベースに対するクエリの構文を定めた言語として SQL^{*10}があります。クエリ言語を用いたデータベースへのクエリは、情報検索（データの抽出・集計）

^{*7} https://en.wikipedia.org/wiki/In-memory_database

^{*8} https://en.wikipedia.org/wiki/Relational_database

^{*9} https://en.wikipedia.org/wiki/Query_language

^{*10} <https://en.wikipedia.org/wiki/SQL>

のみではなく、データの追加、削除、更新処理を含んでいます。

データベースのインデックス (Index^{*11}) は、検索処理を高速化するためのデータ構造であり、インデックス情報を保管するためのストレージスペースと、データ書き込み時のインデックス情報更新のためのリソース消費を伴います。

トランザクション

データベースにおけるトランザクション (Database transaction^{*12}) は、原子性、一貫性、分離性、永続性を備えたデータへの一連の操作をいいます。

*11 https://en.wikipedia.org/wiki/Database_index

*12 https://en.wikipedia.org/wiki/Database_transaction

第3章

Couchbase Server とは 何か

3.1 Couchbase Server の歴史

Couchbase Server の歴史の直接的な始まりは、Membase 社と CouchOne 社の合併により、Couchbase, Inc.^{*1}が設立された 2011 年に遡ります。

Couchbase, Inc. は、Membase 社と CouchOne 社の合併により設立されました。Membase 社は、memcached^{*2}プロジェクトのリーダー達により開発されたキーバリューストア Membase を提供していました。CouchOne 社は、ドキュメント指向データベース Apache CouchDB^{*3}プロジェクトの主要メンバーを擁し、CouchDB の商用サポートを提供していました。

^{*1} <https://www.couchbase.com/about>

^{*2} <https://memcached.org/>

^{*3} <https://couchdb.apache.org/>

インメモリ KVS から JSON データベースへ

2012 年にリリースされた Couchbase Server 2.0 により、Membase を前身とする Couchabse Server は、純粋なキーバリューストア (KVS) から、JSON データベース (ドキュメント指向データベース) へと発展を遂げました。

Couchbase Server は、JSON データを扱うために必要な機能を提供する一方、キーバリューストアとしての側面を残しており、JSON 以外のデータをバリューとして格納することも可能です。また、memcached のように、揮発性のキャッシュとして利用するためのオプションも提供しています (デフォルトではデータをディスクに永続化します)。永続化機構において、メモリファーストアークテクチャーが採用されており、読み込みおよび書き込みのデータ操作において、ミリ秒以下の応答性能を実現しています。

上に説明した Couchbase Server のコア機能は、Data サービスと呼ばれます。なお、Couchbase Server では、リレーションナルデータベースや、他の多くのデータベースでは、「データベース」と呼ばれるデータの格納先を「バケット」と呼びます。

■コラム: 本書におけるサービス名表記方針

Couchbase Server は、Data サービスをはじめとする複数のサービスから構成されています。本書では、一般名詞と明確に区別するため、「Data サービス」のようにサービスの名称には英語表記を用いています。

クエリ

Couchbase Server 4.0 にて、N1QL(Non 1st normal form Query Language) が導入され、Couchbase Server に格納されているデータに対して、SQL 準拠のクエリ言語を用いてアクセスすることができるようになりました。N1QL は、非第一正規形クエリ言語という意味を持ち、ニッケルと発音します。

N1QL クエリは、Query サービスと Index サービスで実現されます。

全文検索

Couchbase Server 5.0 にて、全文検索機能 (Search サービス) が導入されました。Search サービスは、テキストデータに対する単語レベルのインデックス化による全文検索機能に加え、検索語との関連性に応じて文書をスコアリングし検索結果をソートしたり、検索結果中の該当箇所をハイライトするために必要なデータを合わせて提供するといった、アプリケーションに包括的な検索機能を実装するための技術を提供します。

また、Couchbase Server 6.6 では、全文検索機能を N1QL クエリから直接利用することができる Flex インデックスが導入されています。

■コラム: 本書における Search サービスの扱い

本書では、Search サービスについては扱いません。

これは、全文検索は、Couchbase Server 利用において、必ずしも中心的なユースケースとは言えず付加的なものであるのに対して、実践的に使うためには、ドキュメント指向データベースとはまた別の種類のシステムを扱うのに相当する知識が必要とされ、本書の持つ入門書的な性格とはそぐわないためです。

イベント処理

Couchbase Server 5.5 にて、イベント処理機能(Eventing サービス)が導入されました。ユーザーは、「イベント-コンディション-アクション」モデルに基づいてトリガーされる JavaScript を用いたユーザー定義関数を利用できるようになりました。ユーザーは、Couchbase Server 内のデータ更新・削除(イベント)に対して、条件(コンディション)と、それに対するユーザー定義関数(アクション)を定義します。ユーザ定義関数では、Couchbase Server のデータに対する操作のほか、REST API コールによる外部サービス呼び出しを実行することが可能です。

■コラム: エディションによる差異

Eventing サービスは、エンタープライズエディションでのみ利用可能です。

アナリティクス

Couchbase Server 6.0 にて、Analytics サービスが導入されました。この新しいサービスの導入により、Couchbase Server は、OLTP(Online Transaction Processing) と OLAP(Online Analytical Processing) をシングルプラットフォームで実現する HTAP(Hybrid Transaction/Analytical Processing) 環境として利用可能になりました。

Analytics サービスにより、Couchbase Server をバックエンドデータベースとして利用しているビジネスアプリケーションの性能に影響を与えることなく、ニアリアルタイムのデータを用いたアドホックな分析を行うことが可能になります。そのために、OLTP 環境と OLAP 環境を統合する ETL 処

第3章 Couchbase Server とは何か 3.1 Couchbase Server の歴史

理を実装する必要はなく、また、分析用にデータのスキーマを変更する必要もありません。

Analytics サービスは、アドホックな分析要件に適した形でデータを保持します。処理実装には、MPP(Massively Parallel Processing) 技術の最新の成果が用いられており、大量のヒストリカルデータの集計が必要とされるような分析要件に対して、高い応答性能を実現しています。

Analytics サービスが扱うデータは、従来クラスター内部のデータのみでしたが、Couchbase Server 6.6 からは、別の Couchbase Server クラスターのデータ、および S3 ベースのデータストレージにまで拡張されています。

Analytics サービスは、分析エンジン機能のみを担います。UI の実現のために、外部の BI ツールから Analytics サービスへアクセスするための ODBC/JDBC ドライバーが、CDATA^{*4}社から提供されています。

■コラム: エディションによる差異

Analytics サービスは、エンタープライズエディションでのみ利用可能です。

スコープとコレクション

Couchbase Server 7.0において、バケット内のキースペースとして、スコープとコレクションが導入されました。コレクションが導入される前は、Couchbase Server には、RDB におけるテーブルスキーマに相当するものがなかったため、データの種類を表現するためには、ドキュメントの内部にデータの種類に関する情報を持つ必要がありました。

コレクションは、RDB におけるテーブル定義のように、データの構造に

^{*4} <https://www.cdata.com/drivers/couchbase/>

関して制約するものではないため、スキーマレスであることの柔軟性は失われません。

スコープとコレクションの導入により、RDB で管理されているデータを Couchbase Server に移行することが、これまでと比べて、より簡単に行えるようになりました。

運用やセキュリティ機能の拡充

また、Couchbase Server 7.0 では、Backup サービスが導入されました。それまでの Couchbase Server では、バックアップ機能はコマンドラインツールとして提供されていましたが、Backup サービスの導入により、スケジューリングや、Web コンソールからの設定が可能になりました。

ここまで紹介した、新しいユースケースを実現する機能追加だけでなく、運用やセキュリティ等の面においても、Couchbase Server は、その登場以来、進歩を続けています。

■コラム: エディションによる差異

Backup サービスは、エンタープライズエディションでのみ利用可能です。

3.2 NoSQL と現在の技術動向

NoSQL というキーワードが登場してから、長い年月が経ちました。ここでは、NoSQL という技術の現在について整理します。

まずは、現在の情報技術を取り巻く環境について、振り返ることから始めます。

DX (Digital Transformation)

デジタルトランスフォーメーション～DXについて、経済産業省資料「『DX推進指標』とそのガイダンス」⁵では、以下のように定義されています。

企業がビジネス環境の激しい変化に対応しデータとデジタル技術を活用して、顧客や社会のニーズを基に製品やサービス、ビジネスモデルを変革するとともに業務そのものや組織、プロセス、企业文化・風土を変革し競争上の優位性を確立すること

ここで、トランスマーチュン、つまり変革の対象とされているものは2つあります。1つは、企業が生み出し、顧客に提供する製品やサービス、そしてもう1つは、企業内部の業務プロセスです。そして、この変革は、「顧客や社会のニーズを基に」したものであるとされています。

極めて、当たり前のことですが語られているようにみえます。一方で、こう考えることもできるのではないでしょうか、つまり、これまで確かに情報技術によって変革が行われてきた、ただしそれは(十分に)「顧客や社会のニーズ」に基づくものではなかった、と。

SoE (System of Engagement)

SoE、エンゲージメント（のため）のシステム。つまりユーザーとの「つながり」を掲げたシステム。この「SoE」が語られる文脈では、旧来からあるシステムは、SoR（システムオブレコード）、「記録」（のため）のシステム、と呼ばれることになります。

ここで重要なのは、SoRとSoEとの線引きは、「更新系」と「参照系」という古のシステム区分とは異なるということです。「更新系」と「参照系」との関係では、「参照系」システムは、いわば「更新」以外の領域として見出

⁵ <https://www.meti.go.jp/press/2019/07/20190731003/20190731003-1.pdf>

されており、それらはシステムの「機能」から見た区分であるといえます。SoE と SoR との差異は、システムの「目的」、さらにいえば、システムがどのような背景から発想されているか、にあるといえます。

新しい言葉の誕生に立ち会う時、私たちは、その概念が登場する前の風景がどういったものだったか、に思いを巡らすことになります。これまでの（私たちにとって当たり前と思われていた）システムは、（その時点の技術でできること、つまり）システムにとって得意なことから発想されており、ユーザー（とのエンゲージメント）から発想されたものでなかった、そう考えることも出来そうです。

NoSQL の登場まで

NoSQL 登場以前には、データベースの分類といった時、RDB と DWH（データウェアハウス）という区分が、最も定番といえるものでした。そしてそれは、基盤系と情報系というシステムの分類に対応していました。また、OLTP（OnLine Transaction Processing）と OLAP（OnLine Analytical Processing）という用語を、ここで思い起こしても良いでしょう。

NoSQL（NO または、Not Only SQL）という言葉の登場により、データベースの分類に、混乱が招き入れられました。実際「従来のものと異なる」という表現は、カテゴリーとして意味のあるものではなく、純粋なバズワードとして機能することしかできないのは当然でした。乱暴に言えば、NoSQL は、「ビッグデータ」との組み合わせによる類推から（蓄えられたヒストリカルデータの利用という観点から）、広い意味で「情報系」というカテゴリーに投げ入れられることになったとさえいえます。その意味で、結局「Nice To Have」であり、いたずらに DWH よりも複雑なもの、という印象に終始したとさえいえるでしょう（ここでの表現は、極端な断言に聞こえることは承知していますが、ある特定の文脈の提示を意図してのものです）。

SoE と NoSQL

ここで、DX の隆盛を鑑みつつ、SoE のひそみにならって、エンゲージメントのためのデータベース、について考えてみたいと思います。

エンゲージメントのためのシステムにふさわしいデータプラットフォームとはどのようなものでしょうか？ SoE と SoR の境界を考えてみる時、例えば、ネットショッピングにおけるユーザーの行動という視点から次のようにイメージすることができます。

- SoR の範囲: 商品購買・決済
- SoE の範囲: 商品の検索、推薦、カート保管、レビューなど

上記のイメージを援用すれば、ネットショッピングという限定を外して考えても、多くのアプリケーションが SoE の範囲に含まれることがわかるのではないかと思います。

ここで、SoE と SoR の間のシステムの性格の違いを、以下の 3 点に整理してみたいと思います。

- **ユーザーの規模** システムを利用するユーザーの規模は、SoR の非ではない。さらにシステムは、増え続けるユーザー数・利用量に対応していく必要がある。
- **サービスの柔軟性** 新しいサービスを素早くリリースすることが重要であり、また、サービスは常に環境、つまりユーザーとの関係の変化に応じて更新されなければならない。
- **要求される応答速度** サービスの多様化に合わせて、システムは複雑になり続ける。それに応じて、システムの各構成要素、例えば、データプラットフォームの応答速度として許容される時間は短くなる。

そして、これらは「NoSQL」と呼ばれるデータベースの以下の技術要素

にそれぞれ対応している、と言えます。

- 分散アーキテクチャー
- スキーマレス
- インメモリ

以下、それぞれの技術要素について、順に見ていきます。

分散アーキテクチャー

多くの NoSQL(および NewSQL) データベースが共通して分散アーキテクチャーという特徴をもっています。それらのデータベースは、従来のモノリシックなアーキテクチャーからなるデータベースとは異なり、スケールアップによるリソース増強のみではなく、スケールアウトモデルによって、必要に応じた、サービスを停止する必要のない、柔軟なリソースの拡張(あるいは縮小)が可能です。

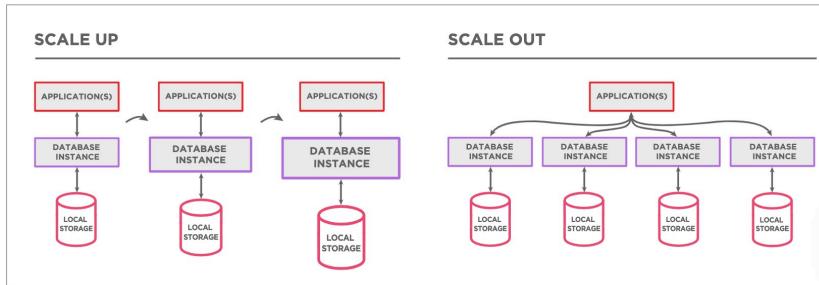


図 3.1: スケールアップモデルとスケールアウトモデル

(画像は、 Moving from Relational to NoSQL: How to Get Started^{*6} より)

^{*6} <https://resources.couchbase.com/c/relational-no-sql-wp?x=Y7B0ca>

り引用)

スキーマレス

スキーマレスという特徴も、多くの NoSQL に共通しています。任意のキーを用いてデータを格納するキーバリューストア（例えば、Redis）と、列指向データベース（例えば HBase、Cassandra）、JSON というツリー構造のデータ記述フォーマットを用いるドキュメント指向データベース（例えば、MongoDB）は、全て NoSQL に分類されますが、スキーマレスという共通の特徴を持ちます。データ構造の柔軟性についていえば、ドキュメント指向データベースには、ツリー構造を表現でき、アプリケーションのデータ（ビジネスドメインオブジェクト）をそのまま格納することができる、という他のデータベースにはない利点があります。

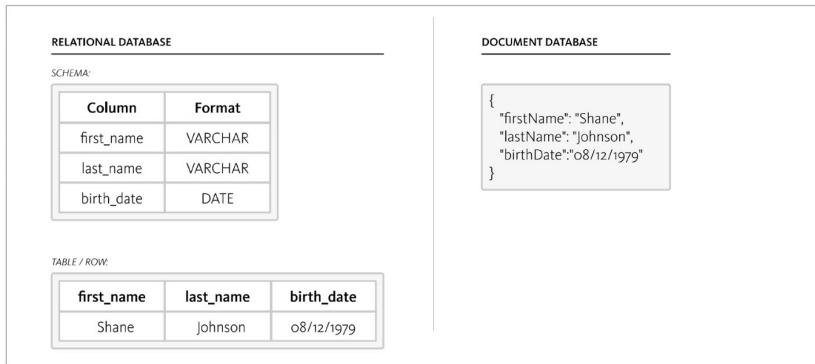


図 3.2: リレーションナルデータベースのテーブルデータ構造とドキュメント指向データベースの JSON データフォーマット

(画像は、 Moving from Relational to NoSQL: How to Get Started⁷ より

⁷ <https://resources.couchbase.com/c/relational-no-sql-wp?x=Y7B0ca>

り引用)

インメモリ

インメモリについては、オープンソースの NoSQL としては、Redis が代表的と言えます。一般には、インメモリデータベースは、(単体でアプリケーションのキャッシュとして用いるのでなければ) 永続化層を担う他のデータベースと組み合わせて利用することによって、高い応答性能という特性を持つデータプラットフォームを実現することができます。

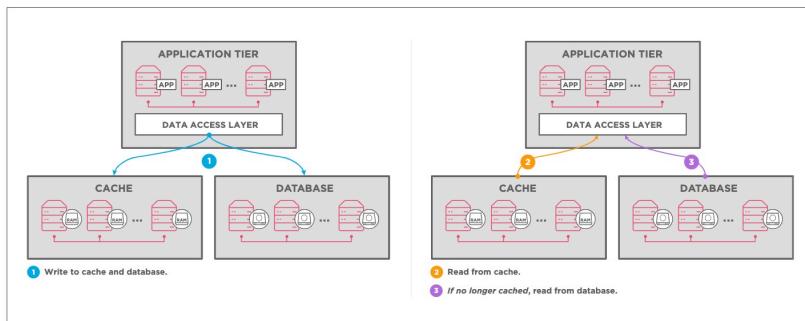


図 3.3: キャッシュとデータベースの併用によるデータアクセスレイヤー実現例

(画像は、 High Performance with Distributed Caching^{*8} より引用)

SoE のためのデータベースとしての Couchbase Server

Couchbase Server は、上で述べてきた 3 つの技術的要素を全て兼ね備えており、エンゲージメントのためのデータベースとして相応しい特徴を持っ

^{*8} <https://resources.couchbase.com/c/high-performance-wp?x=s9hNYZ##>

第3章 Couchbase Server とは 何がメモリーファーストアーキテクチャー

ている、といえます。

NoSQL にカテゴライズされるデータベースには、それぞれ、ビッグデータの集計処理や、リアルタイムかつ大量データの投入等のように、異なった典型的なユースケースがあります。Couchbase Server は、データに対して、リード/ライトの両方のアクセスを、低遅延で実現する必要のある（つまり、インタラクティブな）アプリケーションのバックエンドとして用いられるために設計（最適化）されており、エンゲージメントのための（システムのバックエンド）データベースとして相応しいといえます。

3.3 メモリーファーストアーキテクチャー

インメモリ DB と永続化

ここでは、Couchbase Server のメモリーファーストアーキテクチャーについて理解するため、まずインメモリ DB という技術一般について検討します。また、その具体的な実現として Oracle Coherence と Redis を取り上げます。これらは、いずれもインメモリ分散 KVS(キーバリューストア)というカテゴリーに入ります。また、どちらもコミュニティエディションの存在する OSS ベースのソフトウェアです。

インメモリ DB は、その名の通りデータをメモリに格納します。アプリケーション（データベース利用者）からみたインメモリ DB の性能面の特徴としては、永続化装置の性能がボトルネックにならないデータベースであると言えます（もっとも、インメモリ DB はそもそも永続化の機構を担っていないため、これ自体はさほど意味のない表現ですが）。このデータベースとしての特徴を、永続化を担う他のシステムと組み合わせることによって、従来のデータベースとは異なるレベルの高い応答性能を実現するアーキテクチャーを構築することができます。

第3章 Couchbase Server とは何かメモリーファーストアーキテクチャー

データ永続化パターン

インメモリ DB を、単にアプリケーション層のキャッシュを実現するだけの「分散キャッシュ」と呼ばれるものと区別して考える場合、データの永続化という課題が浮かび上がります。

ここで、データ永続化には以下のような、よく知られたパターンがあります。

- **キャッシュアサイド** アプリケーションは、データへの要求を受けた際に、データがキャッシュの中にあるか調べ、なければデータベースからロードし、将来の別の要求のためにキャッシュの中に保存する。キャッシュアサイドのパターンでは、専用のキャッシュ層が設けられておらず、アプリケーション自身がキャッシュに関連する操作を行う。
- **リードスルー** キャッシュ層が、アプリケーションからデータの読み込み要求を受けた際に、そのデータがキャッシュの中にあるかどうかを調べ、なければデータベースからロードし、将来の別の要求のためにキャッシュの中に保存する。この構成により、キャッシュ層における最適化（例えばデータのプリフェッチ）も可能になる。
- **ライトスルー** キャッシュ層が、アプリケーションからデータへの書き込み要求を受けた際に、データベースへの書き込みと、キャッシュの更新を同時に行う。
- **ライトビハインド** キャッシュ層が、アプリケーションからデータへの書き込み要求を受けるのはライトスルーと同様だが、キャッシュ層にキュー/スレッドを備え、アプリケーションとキャッシュ層との関係（レスポンスタイム、スケーラビリティ）と、キャッシュ層とデータベースとの関係を分離する。この関係の分離により、同一エントリ

第3章 Couchbase Server とは何かメモリーファーストアーキテクチャー

への複数の更新がバッチとしてまとめられる、データベースの障害との直接の影響関係を取り除く、などの利点が生まれる。

データ復旧アプローチ

データ永続化の課題をキャッシュ層における障害発生時のデータ復旧のためのバックアップという観点で捉えることもでき、以下のようなアプローチが存在します（本節の全体の論旨の上では傍系的な話題となります、網羅的にテーマを概観するため整理します）。

- **スナップショット** 定点におけるデータをバックアップファイルとして保存する機能。
- **書き込みログ** RDB におけるいわゆる WAL/REDO ログ。Redis では、AOF(Append Only File) と呼ばれる。

Oracle Coherence と Redis の永続化手法

冒頭にあげた、2つのインメモリ DB について、永続化手法との関係を整理します。

- **Oracle Coherence^{*9}**では、上記のデータ永続化パターン（リード/ライトスルー、ライトビハインド）を想定した実装手法が確立されています。永続化層としては Oracle Database が用いられます。
- **Redis^{*10}**をキャッシュとして使い、ドキュメント指向データベースである MongoDB と連携することによって、永続化を実現する手法

^{*9} <https://coherence.community/>

^{*10} <https://redis.io/>

第3章 Couchbase Server とは何かメモリーファーストアーキテクチャー

が見られます。連携には、CData 社の CData Sync^{*11}を利用するこ
とができます。

永続化機構を備えたインメモリ DB としての Couchbase Server

これまで見たように、インメモリ DB を利用する場合、通常データの永続化は、ユーザーの選択した方法によるインテグレーションにより実現されます。Couchbase Server のユニークなところは、メモリ「ファースト」アーキテクチャーの採用にあります。ユーザーは、メモリ上のデータへのアクセス（ディスク利用による性能制約からの開放）による応答性能を享受しながら、データは、あたかもごく一般的なデータベースのように、ディスク上に保存されている状態が、シングルプラットフォームで実現されています（ここで、「シングルプラットフォーム」という表現は、このようなキャッシュと永続化の両方を満たすための典型的なアーキテクチャーとして、キャッシュレイヤーと永続化レイヤーを2つの異なるテクノロジーによって実現するケースとの対照として用いられています）。

Couchbase Server の内部的な実装は、いわば上述の「ライトビハインド」のパターンに相当します。さらに、一貫性 (Durability) 要件に応じ、永続化装置（および複製）に対して、同期的に更新するオプションを提供しています。オプションの選択は API コール単位であり、システム全体で、ライトスルーかライトビハインドかを固定する必要はありません。

以上、説明したアーキテクチャーのイメージを示します。

^{*11} <https://www.cdata.com/jp/kb/tech/redis-sync-mongodb.rst>

第3章 Couchbase Server とは何かメモリーファーストアーキテクチャー

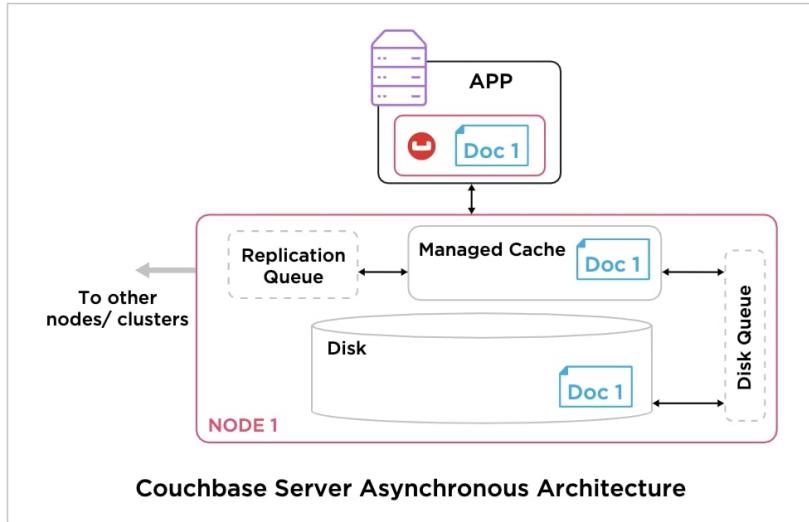


図 3.4: Couchbase Server のメモリーファーストアーキテクチャー

(画像は、Couchbase Under the Hood: An Architectural Overview^{*12} より引用)

■コラム: 純粹なキャッシュとしての Couchbase Server

Couchbase Server は、Memcached プロジェクトのメンバーにより開発された Membase をその前身としており、Memcached のユースケースへの適応が意識されています。こうした背景から、メモリのみを利用して永続化要件のないデータを扱うこともサポートされています。

本書では、Couchbase Server のこの側面については、本来的なデータベースとは異なるユースケースとして、最小限の扱いに留めています。

^{*12} https://resources.couchbase.com/c/server-arc-overview?x=V3nd_e

3.4 JSON データベース

ここでは、なぜ JSON(データベース) が、現在のアプリケーション開発において、重要なのかを見ていきます。

JSON の特徴

まず、先行する技術である XML との比較から、JSON の特徴を整理します。

- **XML** ユーザーが定義したタグを用いて文章構造を記述するマークアップ言語。
- **JSON** キーと値(文字列、数値、真偽値、オブジェクト、およびそれらの配列)のペアの集まり(オブジェクト)によって、ユーザーが定義した構造を持つデータを記述するデータ記述言語。

JSON と XML に共通してみられる特徴として、以下があります。

- データ記述言語であり、データと同時に、データの意味と構造を記述することができる。
- テキストベースのデータ交換用フォーマットである。
- データ構造として、ツリー構造を持つ。ツリー構造は、ネットワーク構造以外のほとんどのデータ表現のニーズに対応できる(RDBで用いられているテーブル構造や、キーと値のペア等、より単純なデータ構造は、ツリー構造に包含される)。
- プログラミング言語を問わず利用できる。
- 仕様が固まっており、十分な歴史・実績を持つ。

さらに、XML に対する JSON のメリットとして、以下があります。

- XML よりも軽量。

- プログラミング言語との親和性が高い。JSON は、JavaScript Object Notation の略記であることが示すように、構文は JavaScript におけるオブジェクトの表記法に由来しており、その他の多くのプログラミング言語が共通して持つデータ構造とも直接的な対応関係を持っている。

JSON と Web フロントエンド技術との親和性

XML が、そして後には JSON が、広く浸透した理由として、その「テキストベースのデータ交換用フォーマット」という特徴が、Web 技術と親和性が高かったことが挙げられます。

「データ交換」のユースケースの中でも、特に重要なのが、Web ブラウザ上での扱いだったといえます。Web ブラウザ上で実行可能なスクリプト言語の仕様として登場した、非同期更新技術 (Ajax^{*13}) は、Web アプリケーションに、ネイティブアプリケーションと遜色のない、ユーザー体験を提供することを実現しました。この非同期更新技術のデータ交換フォーマットとして用いられたのが、当初は XML であり、後には JSON が、実質的に取って代わることになります。

その結果が、現在の SPA(Single Page Application)^{*14} や PWA(Progressive Web Application)^{*15} の隆盛につながっています。そして、このような新しい UI 技術がなぜ重要なと言えば、アプリケーションのユーザビリティの向上によりページビューの増加や離脱率の減少に繋がるからに他なりません。^{*16}

*13 [https://en.wikipedia.org/wiki/Ajax_\(programming\)](https://en.wikipedia.org/wiki/Ajax_(programming))

*14 https://en.wikipedia.org/wiki/Single-page_application

*15 https://en.wikipedia.org/wiki/Progressive_web_application

*16 ディーエムソリューションズ株式会社コラム「PWA (Progressive Web Apps) とは? メリットと実装事例について」(<https://digital-marketing.jp/seo/what-is-progressive-web-apps/>) では、新しい UI 技術 (SPA, PWA) が、ページビューの増加や離脱率の減少へ与える影響について、具体的な数字を交えて紹介されています。

Web アプリケーション開発と高い親和性を持つ Couchbase Server

Web ブラウザー上で実行可能なスクリプト言語として登場した JavaScript は、サーバーサイドでも用いられるようになっています。

一方、JSON は、XML においては広く浸透したとは言い難いデータ管理の目的においても、そのデータ記述言語としての（これまでのデータ管理の主流であった RDB に遙かに勝る）柔軟性により、多くのデータベースにおいて採用されることになりました。

JSON データを扱うドキュメント指向データベースである Couchbase Server を用いることにより、Web フロントエンドとデータベースの両方で、更にはサーバーサイドを含めて、JavaScript/JSON を使ったアプリケーション開発が可能になります。

3.5 トランザクションと NewSQL

ここでは、データベース技術の重要なトピックであるトランザクションについて、データベース技術者が NoSQL(ドキュメント指向データベース)に触れる際に有益と思われる論点を整理しながら、歴史的な展望と併せて、見ていきます。

ドキュメント指向データベースにおけるトランザクションの原子性

ドキュメント指向データベースでは、トランザクションの原子性 (Atomicity) は、ドキュメントという単位と一致しているということができます。

Couchbase Server のドキュメント更新に対するアプローチとして、「get-and-lock API」で実現される悲観的 (ペシミスティック) ロックと「check-

第3章 Couchbase Server とは何か 3.5 トランザクションと NewSQL

and-set (CAS) API』による楽観的（オプティミスティック）ロックの2つがあります^{*17}。

マルチドキュメントトランザクションの必要性

複数のデータ（エンティティ）の関係を、外部参照キーによって表現することは、ドキュメント指向データベースであっても、採用し得る設計です。そのことを踏まえ、複数ドキュメントに跨がるトランザクションが必要なケースとして、次の2つについて考えてみます。

- 複数の（関係のない）エンティティ間でのデータの受け渡し（例えば、口座間での送金）
- 複数の（関係した）エンティティの同時更新（例えば、受注レコードと受注明細）

後者について言えば、ドキュメント指向データベースが扱う準構造化データでは、第一正規化が必須とされないため、必要なトランザクション境界を、1ドキュメントに含まれるデータの範囲と一致させる設計が可能です。

その一方で、NoSQLが従来想定されていた範囲を超えて有益であることが認められてくるに連れて、高度なトランザクションの実現に対する要望が高くなってきました。その結果、NoSQLデータベースが持っていたトランザクションに関する制約を、データベース外部で解決するという試みも行われてきました。Couchbase Serverについてのそのような例として、NECのInfoFrame Relational Store(IRS)^{*18}があります。

そうした時期を経て、昨今ではNoSQLデータベース自体に、マルチドキュメントトランザクションが実装されてきており、その背景には、NoSQL

^{*17} ブログ記事「Optimistic vs Pessimistic Locking - Which One Should You Pick?」(<https://blog.couchbase.com/optimistic-or-pessimistic-locking-which-one-should-you-pick/>)にて、これらの違いと適用基準が解説されています。

^{*18} https://jpn.nec.com/infoframe/relationalstore/docs/irs_couchbase_live_tokyo_2015.pdf

第3章 Couchbase Server とは何か 3.5 トランザクションと NewSQL

のユースケースの範囲が、大きく変わってきてていることが示唆されています。

Couchbase Server の分散トランザクション

Couchbase Server のマルチドキュメントトランザクションは、複数ドキュメントに跨がるトランザクションを実現します。^{*19} Couchbase Server のドキュメントは、物理的に複数のノードに跨がって管理されるので、当然ながら、トランザクションも複数ノードに跨がるものになります。データ管理上の論理的側面における「マルチドキュメント ACID トランザクション (Multi-Document ACID Transactions^{*20})」は、分散アーキテクチャーの観点からの表現である「分散トランザクション」と並列的に成立しており、「分散マルチドキュメント ACID トランザクション (Distributed Multi-Document ACID Transactions^{*21})」という表現も用いられます。

以下に、トランザクションが、複数のノードにまたがって実行されるイメージを示します。

^{*19} <https://docs.couchbase.com/server/current/learn/data/transactions.html#distributed-transactions-multi-node-and-multi-bucket>

^{*20} <https://blog.couchbase.com/couchbase-brings-distributed-multi-document-acid-transactions-to-nosql/>

^{*21} <https://blog.couchbase.com/distributed-multi-document-acid-transactions-in-couchbase/>

第3章 Couchbase Server とは何か 3.5 トランザクションと NewSQL

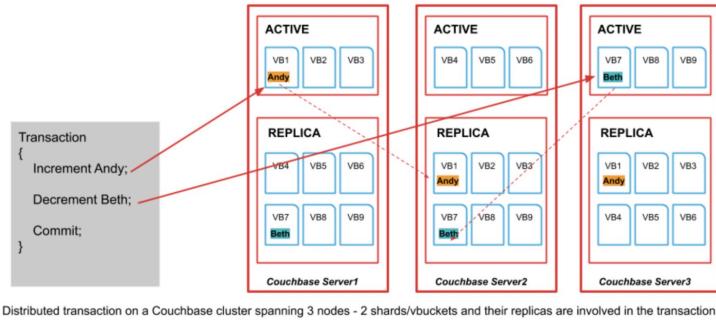


図 3.5: Couchbase Server の分散トランザクション

(画像は、Couchbase Under the Hood: An Architectural Overview^{*22} より引用)

ここで注意しなければならないのは、Couchbase Server のマルチドキュメントトランザクションは、楽観的平行性制御に基づくものであって、上で触れた単一ドキュメント更新の場合と異なり、悲観的ロックを選択することはできないということです。

NewSQL

NewSQL と呼ばれる新しいデータベースの登場は、現在における NoSQL の位置づけを見極める上でも、興味深いものだといえます。その意味で、451 Group^{*23}により作成された以下の図を引用することは、意義深いと思われます。

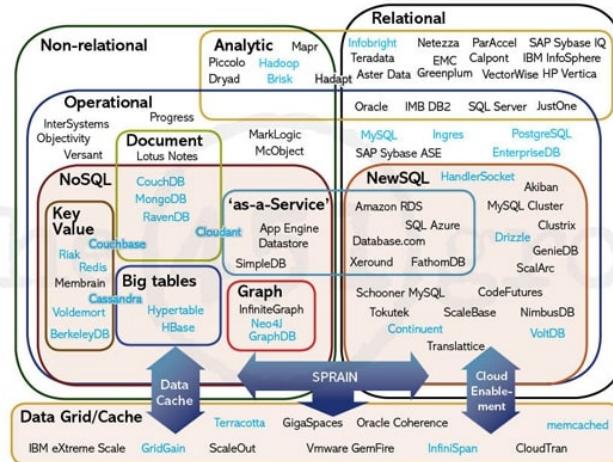
図の中で、Couchbase Server は「Document」と「KeyValue」の両方の

*22 https://resources.couchbase.com/c/server-arc-overview?x=V3nd_e

*23 https://en.wikipedia.org/wiki/451_Group

第3章 Couchbase Server とは何か 3.5 トランザクションと NewSQL

エリアに跨がる場所に位置づけられています。



(画像は、NoSQL, NewSQL and Beyond^{*24} より引用)

NewSQLについての詳細な説明は、上記図の引用元や、その他の文献に譲りたいと思いますが、端的にいえば、Google Spanner^{*25}とそのクローンを差すものと考えて良いかと思います（ここでは、その解釈とします）。その特徴について要約すれば、分散アーキテクチャ（のメリット）を持ち（NoSQLから引継ぎ）ながら、あくまでRDBであること（つまり、トランザクションとSQLを備えていること）を打ち出しているように見えます。例えば、Spannerについては、「Fully managed relational database with unlimited scale, strong consistency, and up to 99.999% availability.」^{*26}と

*24 <https://www.infoq.com/news/2011/04/newsq1/>

*25 <https://research.google/pubs/pub39966/>

*26 <https://cloud.google.com/spanner>

第3章 Couchbase Server とは何か 3.5 トランザクションと NewSQL

謳われています。クローンごとに打ち出し方が異なっているのが見てとれ、CouchbaseDB^{*27}では、「strong consistency」という技術的特徴を「mission-critical」という目的の形で表現している一方、TiDB^{*28}は、「distributed SQL database for elastic scale and real-time analytics」という表現を用いています。YugaByteDB^{*29}は、「Single-Digit Millisecond Latency」を筆頭に挙げているのも興味深いところです。

トランザクションという点についていえば、Couchbase Server のマルチキューメントトランザクションが、リードコミッティドの分離 (Isolation^{*30}) レベルに基づいている一方、NewSQL のトランザクションは、より厳格 (シリアルライザブル) であることが大きく異なっています。これは、SQL が使えることと並び、NewSQL が、(新しい)SQL(/RDB) を標榜している背景にある、重要な特徴と言えるでしょう。Couchbase Server との比較で言えば、SQL(拡張) クエリが使えるところは共通している一方で、明らかに異なる特徴が、このトランザクションの実装ということになります。

楽観的ロックの何が問題か

楽観的ロックの何が問題か、以下の二点に集約して考えてみたいと思います。

- トランザクション競合発生時の対応がリトライに基づくため、最終的に（集中時には）性能の問題に帰結する場合がある。
- 悲観的ロックに基づくデータベースでの開発を常識としてきた開発者にとって、これまでの流儀を全く変えずに開発ができるわけではない。

^{*27} <https://www.cockroachlabs.com/>

^{*28} <https://pingcap.com/>

^{*29} <https://www.yugabyte.com/>

^{*30} [https://en.wikipedia.org/wiki/Isolation_\(database_systems\)](https://en.wikipedia.org/wiki/Isolation_(database_systems))

第3章 Couchbase Server とは何か 3.5 トランザクションと NewSQL

高いトランザクションレベル（悲観的ロック）を取ることが絶対に重要で、システムに必要な要件であるユースケースがある一方（その場合、メリット・デメリットを比較する必要えないため、ここでは必ずしもそのことを前提としないケースについて検討します）、前者の欠点は、アプリケーション全体において、トランザクション/ロックの要件が必要である場合には、それ自体単体で見れば悲観的ロックと比較して軽量な処理であるはずの楽観的ロックの実装が問題となりうることを示しています。後者については、一概に結論付けられるようなものではなく、アーキテクチャー的なアドバンテージなど諸々を総合的に考慮した上で、判断が必要になると思われますが、よく知られた、少なくない数のリレーションナルデータベースで、デフォルトの分離レベルとしてリードコミッティッドが採用されているのも事実です。

性能を犠牲にしないトランザクション適用の柔軟性を持つ Couchbase Server

NoSQLでは、当初、RDBのような複数レコードに跨がるトランザクションはサポートされていませんでした。上に見てきたように、今となっては、それも昔の話です。一方、後天的にトランザクションが実装されたデータベースに対して度々見られる批判として、当初有していた性能面の利点が捨て去られ、データベースとして凡庸な位置づけに墮した、というものがあります。トランザクションの実現とデータベースの性能は、論理的にトレードオフの関係であることは明白ですが、Couchbase Serverでは、リクエストベースで、トランザクションの適用を選択することができ、必要な箇所でのみトランザクションの恩恵を受けることができます。

■コラム: 関連情報紹介「How we implemented Distributed Multi-document ACID Transactions...」

より詳しい内容に興味がある方へ向けて、ブログ記事「How we imple-

mented Distributed Multi-document ACID Transactions in Couchbase」^{*31}を紹介します。

この記事では、RDBMS と NoSQL(ドキュメント指向データベース)、そして NewSQL のトランザクションを比較した上で、Couchbase Server におけるマルチドキュメント分散 ACID トランザクションがどのように実装されているかを解説しています。

結論部から、Couchbase Server のトランザクションの優位性を列挙します。

- Low Cost – Pay for what you use
- No Central Transaction Manager or Central Coordinator
- No Internal Global Clock and no MVCC(Multiversion Concurrency Control)
- Flexibility

最後の「Flexibility」については、データベースによって一律に同じトランザクション管理方法が強制されるのではなく、アプリケーションの必要性に応じて、記事中で紹介されている様々な機能を使い分けることができるこことを指しています。

3.6 第一正規化制約を超えて

NoSQL のカテゴリーについて

NoSQL データベースの世界では、JSON を扱うデータベースは、「ドキュメント指向データベース（ドキュメントストア）」と呼び習わされてきました。

^{*13} <https://blog.couchbase.com/distributed-multi-document-acid-transactions/>

た。「ドキュメント」は普通名詞としてあまりに一般的すぎるため、「準構造化データベース」とでも呼んだ方が、実態をよりよく表現する、ということができます。言い換えると、「ドキュメント指向データベースはドキュメント形式のデータを扱うが、ドキュメント形式のデータとは、JSON や XML のような準構造化データである」という定義よりもよほどシンプルだと言えます。

「準(半)構造化データ」という呼称について

「semi-structured data」という用語には、暗に structured(構造) イコード、リレーションナルデータベースのテーブル形式、ということが含意されている、と解釈できそうです。「構造」という言葉を素直に受け止めるのであれば、これはリレーションナルデータベースに対する行き過ぎた偏向であって、テーブル形式以外の方法でもデータは構造化することができ、JSON データは、その意味で紛れもなく「構造」を持っているといえます。

上記でいうところの「テーブル形式」は、リレーションナルデータベースにデータを格納する際の制約である、第一正規化が行われた構造、ということができ、翻って JSON データについて、「準構造化データ」というよりも、「非第一正規形データ」と呼ぶ方が、より具体的な内実を表しているとも言えそうです。

Couchbase Server と RDB のデータモデルの違いと共通点

Couchbase Server には、スコープとコレクションという、RDB におけるスキーマとテーブルに相当する概念が存在します。Couchbase Server のコレクションは、RDB におけるテーブルのように、データの種類(エンティティ)を表現することができますが、RDB におけるテーブルとは異なり、データ構造を制約するものではありません。

そして、RDB のテーブルに格納されるデータの条件である、第一正規形

データは、準(半)構造化データに包含されます。

このように、Couchbase Server は、データのエンティティに関する表現(コレクション)を持ち、テーブル(表)形式のデータを難なく扱えることから、RDB のデータ(モデル)を Couchbase Server に移行(応用)することは、造作なく行えます。そして、この関係は非可逆であると言えます。つまり、JSON は第一正規形データを表現することができ、また、スキーマレスであることは静的なスキーマ定義を問題視しませんが、その逆は成り立ちません。こうした点を踏まえ、「RDB のスーパーセットとしての Couchbase Server」について考えてみることができます。

RDB のスーパーセットとしての Couchbase Server

Couchbase Server のクエリ言語には、N1QL (Non 1st normal form Query Language) という名称が付けられています。N1QL では、SQL をベースに、JSON 固有の要素、配列やネストされたデータ構造に対して、構文が拡張されています。視点を変えれば、JSON 固有の要素が介在しない場合、RDB で使われているものと同じクエリを使うことができるといえます。

そうした面から見ると、先に検討したような JSON の表現力の点のみではなく(それだけであれば、他のドキュメント指向データベースについても同じことが言える訳ですが)、N1QL を備えているからこそ、Couchbase Server は RDB のスーパーセット足り得ている、という見方が成立します。この SQL が使えるという利点は、NoSQL データベースとしては例外的ですが、NoSQL 以降に登場した NewSQL データベースには備わっている特徴です。そうした面から見ても、Couchbase Server(のデータベースとしての位置付け)は興味深いと言えるでしょう。

もっとも、扱うデータ型の種類について、JSON は SQL 標準よりも限定されていること、Couchbase Server の複数のドキュメントに跨がるトランザクションのトランザクション分離レベルは、一般の RDB よりも限定期であること、こういった諸々の違いは認識しておく必要があります。にも関わ

らず、「スーパーセット」という挑発的な表現を用いたのには、NoSQL という名称が、RDB へのアンチテーゼとしての意味合いを持ち、NoSQL データベース一般がそうしたものと目されている状況に対して、行き過ぎを正す意図があります。多くの NoSQL データベースが、データ集約基盤やヒストリカルデータ分析等のような RDB とは異なるユースケースで使われているのに対して、Couchbase Server は、インタラクティブなアプリケーションのバックエンドデータベースとして、設計・最適化されており、これは RDB の主要なユースケースと共通しています。

NoSQL の様々なカテゴリーの中でも、ドキュメント指向データベースは、「そのデータ構造の柔軟性を生かしてアジャイル開発に用いられる」^{*32}というケースを典型として、インタラクティブなアプリケーションとの親和性は、元々高かったと言えます。一方で、現在の Couchbase Server が備えている様々な特徴、つまり、メモリーファーストアーキテクチャーが実現するミリ秒以下の応答性能、SQL についての知識と経験を有効に活用できる N1QL クエリ、そしてマルチドキュメントトランザクションが実現する従来の範囲を超えたユースケース等、これらを踏まえて考えると、「ドキュメント形式のデータ」の「データ構造の柔軟性」からの発想は、ミスリーディングにさえなりかねません。^{*33}

言い換えれば、アプリケーションが扱うデータ形式（この場合、JSON であること）からスタートして、採用するデータベースを決定するルートは、必ずしも必然ではない、ということです。JSON のツリー構造は、より単純なデータ構造にも適用できるため、例えばレガシーシステムをモダナイズする場合のように、データ構造の柔軟性が必ずしも必要でなく、アジャイル開発が適用されないプロジェクトを含め、Couchbase Server を活用することのできる場面は多岐に渡って存在します。

^{*32} さらに言えば、「厳格なデータ設計プロセスをオミットして開発を始める」という表現も可能でしょう。

^{*33} 特に、先の脚注に引いた表現と組み合わされる場合には。

第 4 章

Couchbase Server を使つ てみる

4.1 インストール

ダウンロード

Couchbase Server のインストーラーは、下記の URL からダウンロードすることができます。

<https://www.couchbase.com/downloads>

ダウンロード時に、インストール先の OS として Windows、macOS に加え、様々な Linux 系 OS を選択可能です。

インストール手順

各 OS 毎のインストール手順は、ドキュメント^{*1}に詳述されています。

^{*1} <https://docs.couchbase.com/server/current/install/install-intro.html>

Linux 系 OS は、コマンドラインからパッケージシステムを介してインストールすることが可能です。

インストール以降のクラスター構築手順は、OS の別に依らず、共通しています。

4.2 Docker コンテナ利用

初回実行

下記のコマンドを実行します。ここでは、`--name` オプションとして `cb` を指定しており、以降の操作でこの名称を利用します。

```
$ docker run -d --name cb -p 8091-8094:8091-8094 -p 11210:11210 couchdb
```

Couchbase の Docker リポジトリから最新版がダウンロードされ、コンテナの実行が開始されます。

起動が成功すれば、ブラウザで下記の URL にアクセスした際に、「Setup New Cluster」と「Join Existing Cluster」のいずれかを選択する画面が表示されます。

<http://localhost:8091>

ログの確認

起動時のログを確認するには、以下のコマンドを実行します。

```
$ docker logs cb
```

```
Starting Couchbase Server -- Web UI available at http://<ip>:8091  
and logs available in /opt/couchbase/var/lib/couchbase/logs
```

イメージの確認

```
$ docker images  
REPOSITORY          TAG      IMAGE ID  
couchbase           community  21fad80a0f9e
```

プロセスの確認

```
$ docker ps  
CONTAINER ID        IMAGE               COMMAND       CREATED  
21fad80a0f9e        couchbase:community   "/entrypoint.sh couc  
..."    4 weeks ago      Up 2 seconds   8095-8096/tcp, 0.0.0.0:8091-8094->
```

上記の手順により couchbase レポジトリの community というタグを持つ、Docker イメージがインストールされ、そのイメージから cb という名前のコンテナが作られ、稼働中の状態であることが確認されます。

コンテナの停止と再起動

コンテナをストップします。

```
$ docker stop cb
```

ブラウザで Web コンソールにアクセスできないことを確認します。

下記のコマンドを再度実行して、プロセス情報が表示されないことを確認します。

```
$ docker ps
```

実行されていないコンテナの情報を確認するには、以下のようにオプションをつけて実行します。

```
$ docker ps -a
```

コンテナを改めて開始してみます。

```
$ docker start cb
```

Web コンソールにアクセスできることを確認します。

コンテナ上のコマンド実行

コンテナに含まれるコマンド(例えばここでは `cbbbackupmgr` コマンド)を実行する場合、以下のように行います。

```
$ docker exec -it cb /opt/couchbase/bin/cbbackupmgr -h
```

コンテナとイメージの削除

コンテナを削除する場合は、以下のコマンドを実行します（コンテナが起動している場合には、コマンド実行前にコンテナを停止します）。

```
$ docker rm cb
```

ダウンロードされたコンテナイメージを削除する場合は、以下のコマンドを実行します。

```
$ docker rmi couchbase:community
```

4.3 クラスター初回構築

クラスターの構築には、CLI や REST API を用いる方法もありますが、ここでは Web コンソールを使った手順について解説します。

Web コンソールへのアクセス

<http://localhost:8091> にアクセスします。

インストール直後の状態では、以下の様な画面が表示されます。



図 4.1: インストール直後の画面

クラスターセットアップウィザード

インストール直後の画面上の選択肢、「Setup New Cluster」と「Join Existing Cluster」の2つから、「Setup New Cluster」を選択します。次の画面が表示されるので、適宜入力します。

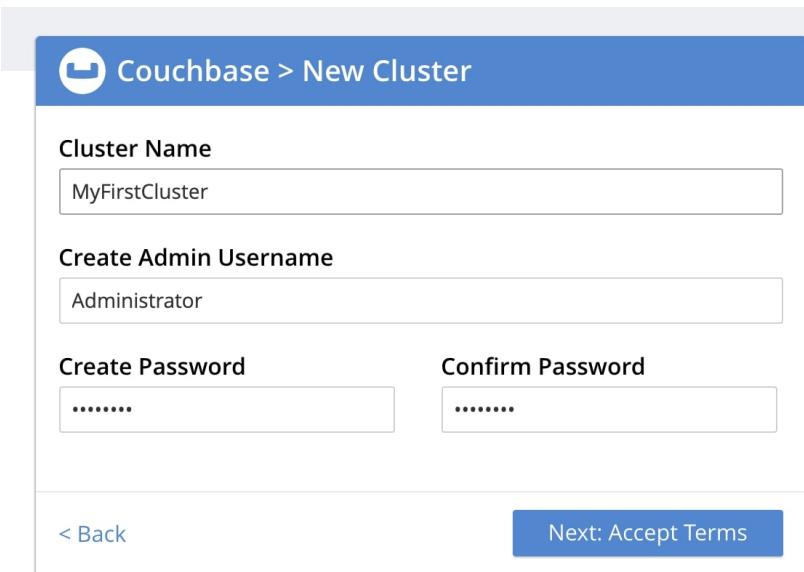


図 4.2: 新規クラスター作成ダイアログ

「Next: Accept Terms」を押下します。
適宜、チェックボックスをチェックします。

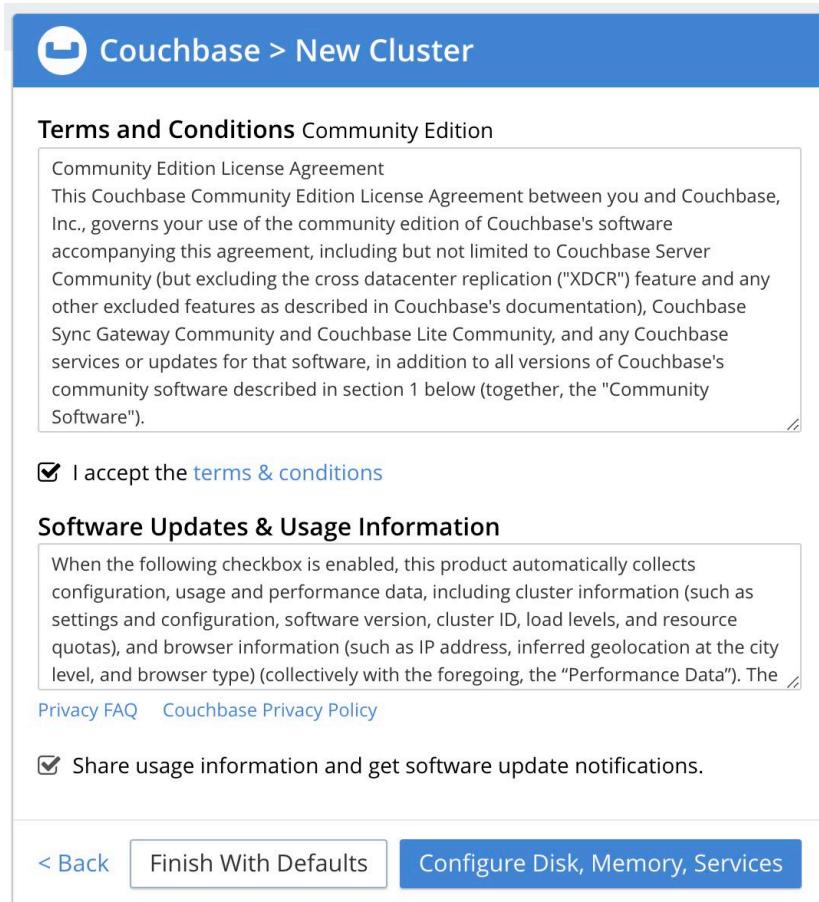


図 4.3: 利用規約ダイアログ

「Configure Disk, Memory, Services」を押下します。

クラスター設定

クラスター設定ダイアログが表示されます。

サービス毎に割り当てるメモリを、「Service Memory Quotas」として設定できることができます。

また、不要なサービスのチェックボックスを外すことにより、このノードで、そのサービスが無効になります。後から有効にすることはできませんが（その場合は、クラスターから一旦削除し、再追加する際に、改めてサービスを選択します）、クラスター全体の構成を柔軟に設計することができます。

ここでは、Search（全文検索）サービスを無効にしています。

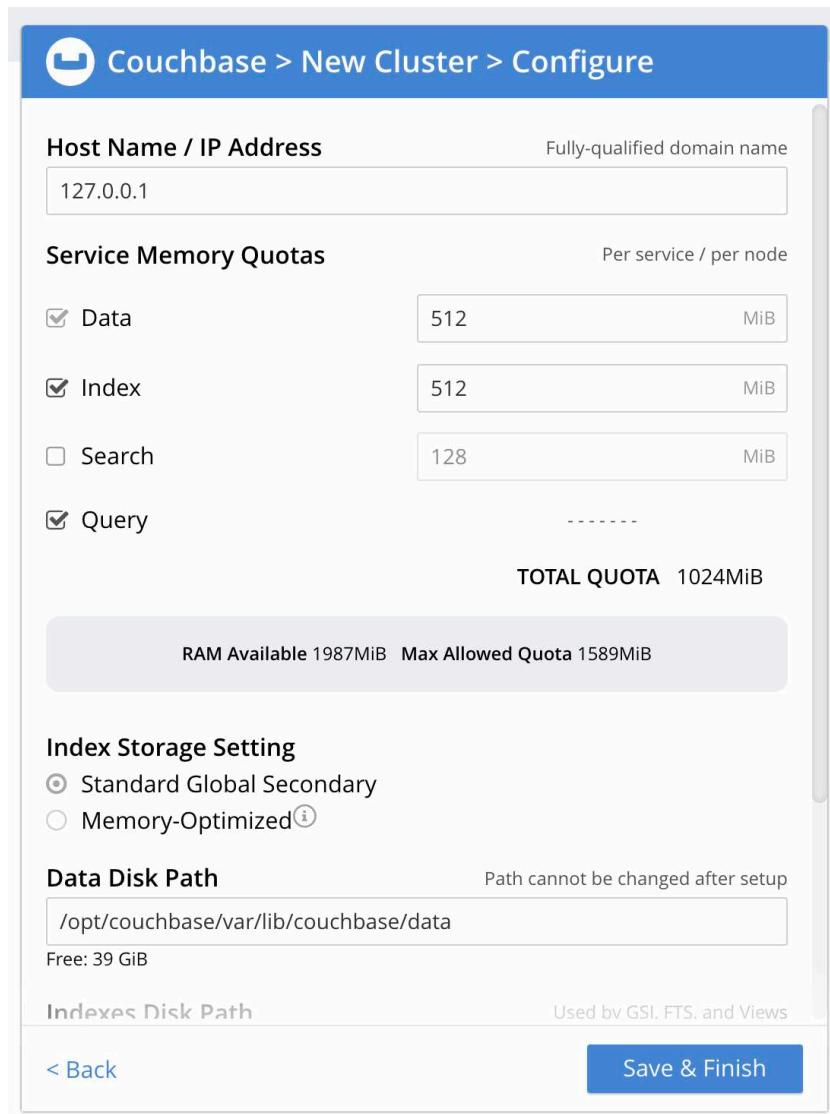


図 4.4: クラスター設定ダイアログ

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方

「Save & Finish」を押下します。

ウィザードが終了し、Web コンソールのダッシュボード初期画面が表示されます。



図 4.5: ダッシュボード初期画面

ここまで手順で、Couchbase Server が 1 ノードで構成されるクラスターとして構築されました。

4.4 Web コンソールの使い方

ここでは、Web コンソールの基本的な使い方を解説します。

バケット作成

まず、他の多くのデータベースにおける「データベース」に相当する、Couchbase Server のデータ格納先であるバケットを作成します。

作成手順は、以下の通りです。

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方

1. サイドメニューから、「Buckets」を選択し、バケット一覧画面を表示します。
2. 画面右上の「ADD BUCKET」リンクをクリックします。バケット追加ダイアログが表示されます。
3. バケット名を入力します。
4. 「Advanced bucket settings」を展開します。ここでは、「Replicas」セクションの「Enable」チェックボックスを外します（レプリカを有効にするには最低2ノード必要であるため、開発や検証のために1ノードでクラスターを構成する場合、チェックボックスを外さずにバケットを作成すると、Web コンソールに常時警告が表示されます）。また、開発時の利便性のため、「Flush」セクションの「Enable」チェックボックスにチェックし、フラッシュ機能を有効にします（フラッシュ機能を有効にすると、バケット一覧画面で「Flush」ボタンが表示され、このボタンを押下する操作で、バケットのデータをすべて削除することができます）。
5. 「Add Bucket」ボタンを押下します。

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方

Add Data Bucket X

Name
user_management

Memory Quota in megabytes per server node
512 MiB

other buckets (0B) this bucket (512MiB) available (0B)

Bucket Type
 Couchbase Memcached Ephemeral

▼ Advanced bucket settings

Replicas
 Enable
 Replicate view indexes

Ejection Method ⓘ
 Value-only Full

Bucket Priority ⓘ
 Default High

Minimum Durability Level ⓘ
none ▾

Auto-Compaction ⓘ
 Override the default auto-compaction settings?

Flush ⓘ
 Enable

59 Cancel Add Bucket

図 4.6: バケット追加ダイアログ

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方

The screenshot shows the 'Buckets' section of the Couchbase Web Console. A single bucket named 'user_management' is listed. The bucket details are as follows:

- Type: Couchbase
- Bucket RAM Quota: 512MB
- Cluster RAM Quota: 512MB
- Replicas: disabled
- Server Nodes: 1
- Ejection Method: Value-Only
- Conflict Resolution: Sequence Number
- Compaction: Not active
- Compression: Off
- Storage Backend: CouchStore
- Minimum Durability Level: none

On the right side, there are tabs for 'Documents' and 'Scopes & Collections'. Below the bucket list, there are four buttons: 'Drop', 'Compact', 'Flush', and 'Edit'.

図 4.7: バケット一覧画面

スコープ作成

作成済みのバケットにスコープを作成する手順は、以下の通りです。

1. サイドメニューから、「Buckets」を選択し、バケット一覧画面を表示します。
2. スコープを追加する対象のバケットの右端の「Scopes & Collections」リンクをクリックします。スコープ一覧画面が表示されます。
3. 画面右上の「ADD SCOPE」リンクをクリックします。スコープ追加ダイアログが表示されます。
4. スコープ名を入力します。
5. 「Save」ボタンを押下します。

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方

The screenshot shows the 'Add Scope' dialog box. At the top, it says 'Add Scope' and has a close button 'X'. Below that, there's a 'Bucket Name' field containing 'user_management' with a dropdown arrow. Underneath is a 'New Scope Name' field containing 'japan'. At the bottom right are 'Cancel' and 'Save' buttons.

図 4.8: スコープ追加ダイアログ

コレクション作成

作成済みのスコープにコレクションを作成する手順は、以下の通りです。

1. サイドメニューから、「Buckets」を選択し、バケット一覧画面を表示します。
2. コレクションを追加する対象のバケットの右端の「Scopes & Collections」リンクをクリックします。スコープ一覧画面が表示されます。
3. コレクションを追加する対象のスコープの右端の「Add Collection」リンクをクリックします。コレクション追加ダイアログが表示されます。
4. コレクション名を入力します。
5. 「Save」ボタンを押下します。

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方



図 4.9: コレクション追加ダイアログ

スコープとコレクションを作成した後の、スコープ一覧画面は、以下のように表示されます。

The screenshot shows the 'Scopes & Collections' page. On the left, there's a sidebar with 'Dashboard', 'Servers', 'Buckets', 'XDCR', 'Security', 'Settings', 'Logs', 'Documents', 'Query', 'Indexes', 'Search', and 'Views'. The 'Buckets' tab is selected. In the main area, there's a table for the 'japan' scope. The table has columns for 'scope name', 'collections', 'items', 'memory used', 'disk utilization', and 'ops/sec'. There are two collections: '_default' and 'users'. The 'users' collection has 0 items and 0B memory used. At the bottom right of the table, there are buttons for 'Drop', 'Documents', and 'Add Collection'. There are also 'filter scopes' and search input fields at the top right of the table. The overall URL in the browser is 'MyFirstCluster > Buckets > Scopes & Collections'.

図 4.10: スコープ一覧画面

ユーザー作成

クラスターにユーザーを追加する手順は、以下の通りです。

1. サイドメニューから、「Security」を選択します。
2. 画面右上の「ADD USER」リンクを押下します。ユーザー追加ダイ

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方

アログが表示されます。

3. 入力欄 (Username,Full Name,Password) に入力します。
4. 画面右半分の「Roles」ツリー画面で、必要な権限を設定します。例えば、「Bucket」の下の「Application Access」で、アプリケーションから利用するバケットを探して、「ADD」ボタンを押下します。
5. 「Add User」ボタンを押下します。

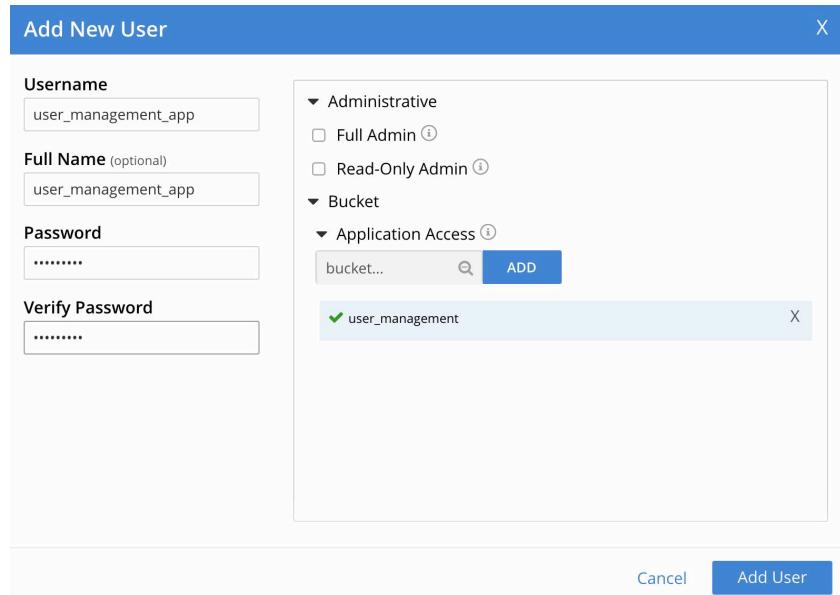


図 4.11: ユーザー追加ダイアログ

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方

The screenshot shows the 'Security' section of the Couchbase Web Console. The top navigation bar includes 'activity', 'help', 'Administrator', 'ADD USER', and tabs for 'Users' and 'Session'. On the left, a sidebar lists 'Dashboard', 'Servers', 'Buckets', 'XDCR', and 'Security' (which is selected). A search bar at the top has 'filter by username...' and a magnifying glass icon. Below it, a table displays user information:

username	roles	auth domain	password set
user_management_app	Application Access [user_management]	Couchbase	9 Sep, 2021

At the bottom, there's a page number '20' and navigation links '< prev | next >'.

図 4.12: ユーザー一覧画面

ファイルからのデータ登録

既存のデータベースからの移行等の目的で、外部ファイルのデータを Couchbase Server へ登録する手順は、以下の通りです。

外部ファイルを用いたデータ登録は、`cbimport` コマンドを用いてコマンドラインから行うこともできます（詳しい操作方法をコマンドライン操作の章で説明しています。Web コンソールから実行する場合の各設定項目のオプションの意味についても、そちらを参照ください）。

1. サイドメニューから、「Documents」を選択します。
2. 上部メニューから、「Import」を選択します。
3. 「Select File to Import...」ボタンを押下し、開いたダイアログで、登録するデータを選択します。
4. データが「File Contents」に表示されます。ファイルの内容に基づいて、必要な設定を行います。ファイル形式（JSON または CSV）とデータを登録するキースペースを指定し、ドキュメントの ID の指定方法を選択します。
5. ここまで入力した内容に従って、`cbimport` コマンド実行用のパラメータが更新されます。この内容を使って、コマンドラインから同じ操作を実行することも可能です。
6. 画面下の「Import Data」ボタンを押下します。

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方

The screenshot shows the 'Import' tab of the Couchbase Web Console. On the left, there's a 'Select File to Import...' button with 'user_data.csv (0 MiB)' selected. Below it, 'Parse File As' is set to 'CSV' and shows 'found 2 records'. A 'Keyspace' dropdown is set to 'user_manag...' with 'japan' selected. Under 'Import With Document ID', the radio button 'Value of Field:' is selected with 'user_id' chosen from a dropdown. The right side shows 'File Contents' with a table:

name	region	user_id
田中	japan	1
佐藤	japan	2

Below the table are tabs for 'Raw File', 'Parsed Table' (which is selected), and 'Parsed JSON'. At the bottom is a blue 'Import Data' button.

図 4.13: ファイルデータ登録画面

サンプルデータ利用

Couchbase Server には、サンプルデータが同梱されています。

サンプルデータを用いて、新しいバケットを作成する手順は、以下の通りです。

1. サイドメニューから、「Settings」を選択します。
2. 上部メニューから、「Sample Buckets」を選択します。
3. 「Available Samples」として示されているバケット名から、利用するデータを選択します。
4. 「Load Sample Data」ボタンを押下します。ロードされたバケットの名前は、「Installed Samples」に移動されます。

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方

The screenshot shows the 'Settings' page of a cluster named 'MyFirstCluster'. On the left, a sidebar lists navigation options: Dashboard, Servers, Buckets, XDCR, Security, Settings (which is selected), and Logs. The main content area is titled 'Sample Buckets' and contains the following text:
Sample buckets contain example data, views, and indexes for your experimentation.
Sample buckets — like all buckets in Couchbase Server 5.0+ — can only be accessed by a user with privileges for that bucket.

Below this, there are two sections: 'Available Samples' and 'Installed Samples'. Under 'Available Samples', there are three checkboxes: beer-sample, gamesim-sample, and travel-sample. Under 'Installed Samples', it says 'none'. At the bottom of this section is a blue button labeled 'Load Sample Data'.

図 4.14: サンプルバケット登録画面

クエリ実行

Web コンソールのクエリワークベンチを用いて、クエリを実行する手順は、以下の通りです。

1. サイドメニューから、「Query」を選択します。
2. クエリワークベンチがデフォルトで表示されます。他の画面に遷移した後は、上部メニューから、「Query Workbench」を選択します。
3. 「Query Editor」に N1QL を入力します。
4. 「Execute」ボタンを押下してクエリを実行します。

第4章 Couchbase Server を使ってみる 4.4 Web コンソールの使い方



The screenshot shows the 'Query' tab selected in the left sidebar. In the main area, a 'Query Editor' window displays the following command:

```
1 CREATE PRIMARY INDEX primary_idx_users ON user_management.japan.users;
```

The 'Execute' button is highlighted in blue. Below the editor, the results pane shows a single line of JSON output:

```
1- {
2   "results": []
3 }
```

The right side of the interface includes a 'Explore Your Data' section with a tree view of collections: 'user_management' (2 collections: '_default', 'japan').

図 4.15: クエリワークベンチ画面

インデックス作成・確認

Web コンソールのクエリワークベンチを用いて、インデックスを作成する手順は、以下の通りです。

1. サイドメニューから、「Query」を選択します。
2. クエリワークベンチがデフォルトで表示されます。他の画面に遷移した後は、上部メニューから、「Query Workbench」を選択します。
3. 「Query Editor」に、CREATE INDEX DDL(データ定義言語)を入力します。
4. 「Execute」ボタンを押下します。

作成されたインデックスを確認するには、以下の手順を踏みます。

1. サイドメニューから、「Indexes」を選択します。
2. 「Bucket & Scope」セクションで、キースペースを選びます。
3. 選択したキースペースに作成済みのインデックスが表示されます。インデックス列の右端の「status」が「ready」になっていれば、インデックスが利用可能な状態です。

index name	requests/sec	resident ratio	items	data size	keyspace	status
primary_idx_users	0	0%	0	31.9KB	user_management.japan.us...	ready

図 4.16: インデックス一覧画面

4.5 クラスター構成変更

ここまで、1ノードからなるクラスターを構築し、Webコンソールでの操作方法を見てきました。本書のサンプルアプリケーションを動かすために、複数ノードからなるクラスターを用いる必要もなければ、N1QLクエリなどの開発に必要な知識の理解において、基本的にノード数の違いを意識するところはありません。

とはいって、クラスターは、複数ノードで構成されることが前提としてあり、1ノードでの構成は、その1つの形態に過ぎないのも確かです。そこで、ここでは既存のクラスターに対して、ノードの追加あるいは削除を行い、クラスターの構成を変更する手順について、概要を記します。

クラスターの構成変更には、CLIやREST APIを用いる方法もありますが、ここではWebコンソールを使った手順を用いて、クラスター構成変更のプロセスを解説します。

ノードの追加

クラスターに新しくノードを追加する場合、まずその追加するノード(サーバー、VMインスタンス、あるいはコンテナ)に、Couchbase Server

がインストールされていることが前提となります。

その上で、ノードの追加方法には以下の2通りがあります。

- クラスターの既存ノード（の Web コンソール）から実行
- クラスターへ追加するノード（の Web コンソール）から実行

クラスターの既存ノードから実行する場合、以下の手順になります。

1. サイドメニューから、「Servers」を選択します。
2. 画面右上の「ADD SERVER」リンクを押下します。
3. 「ADD Server Node」ダイアログで、「Hostname/IP Address」など
の入力項目を入力し、「Add Server」ボタンを押下します。

クラスターへ追加するノードから実行する場合、以下の手順になります。

1. インストール直後のトップ画面の選択肢から、「Join Existing Cluster」を選択します。
2. 「Cluster Host Name/IP Address」、「Cluster Admin Username」、
「Cluster Admin Password」を適宜入力します。
3. 必要に応じ、「Configure Services & Settings For This Node」を開
き、編集します。
4. 「Join With Default Configuration」ボタンを押下します。

既にそのクラスターにバケットが作成されている状況を考えると、追加したノードを含めてバケットを均等に分散する必要があることが分かります。この処理を実行するために、クラスターのリバランスを実施します。

リバランス実施前は、そのノードは「New Node」としてマークされており、まだそのノードではトライフィックが発生していません（この時点では、「Cancel Add」を押下して、追加をキャンセルすることができます）。リバランスを実施するには、Servers 画面右上に表示される「Rebalance」ボタンを押下します。リバランスが完了してはじめて、追加されたノードは、実際にクラスターのメンバーになります。

ノードの削除

1. サイドメニューから、「Servers」を選択します。
2. 削除する対象のサーバーをクリックします。
3. 右下の「Remove」ボタンを押下します。

既にそのクラスターにバケットが作成されている状況を考えると、単にノードをクラスターから取り外してしまうと、バケットが一部分失われることになり適切ではありません。この問題を解消するために、クラスターのリバランスを実施します。

リバランス実施前は、そのノードは「Node flagged for removal」としてマークされており、まだそのノードでトラフィックが発生し続けています(この時点では、「Cancel Remove」を押下して、削除をキャンセルすることができます)。リバランスを実施するには、Servers 画面右上に表示される「Rebalance」ボタンを押下します。リバランスが完了してはじめて、削除されたノードは、実際にクラスターのメンバーではなくなります。

第5章

データ

ここでは、クライアント/アプリケーションからみたデータについて解説します。Couchbase Server では、データは複数のノードに跨って管理されますが、データを利用する場合、表面的には、そのことを意識する必要はありません。

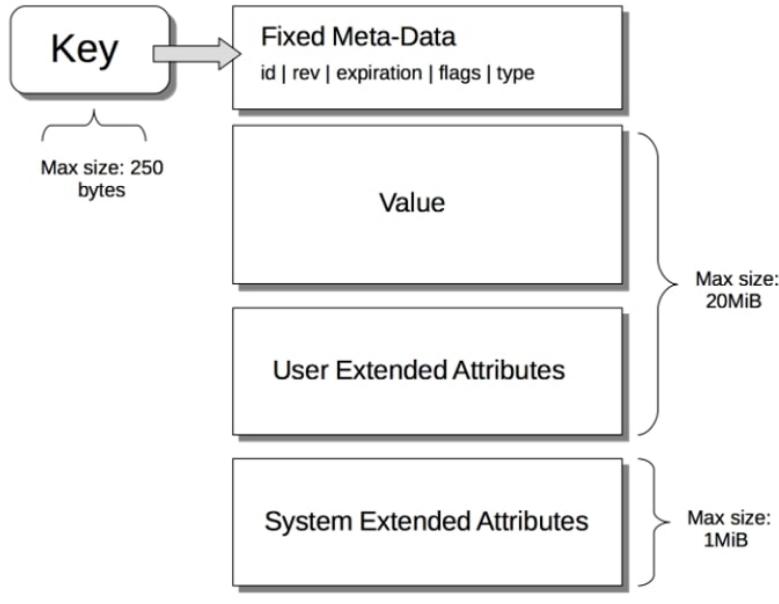
5.1 ドキュメント

概要

Couchbase Server における、データ格納の最小単位はドキュメントです。

ドキュメントは、以下の構成要素からなります。概念図とあわせてご確認ください。

- キー
- 値
- メタデータ
- ユーザー拡張属性
- システム拡張属性



(画像は、Couchbase Server ドキュメント Size Limits^{*1}より引用)

以下、ドキュメントを構成する要素について、基本的な仕様を整理します
(拡張属性については割愛します)。

キー

- 可変長 (最大 250 バイト)
- キースペース内において一意
- スペースを含まない UTF-8 文字列

^{*1} <https://docs.couchbase.com/server/current/learn/data/data.html#size-limits>

値

- 可変長 (最大 20MB)
- 値を JSON(RFC 8259^{*2}に準拠) として持つことによって、クエリ等のドキュメント指向データベースとしての各種機能を活用することができるが、バイナリデータを保存して KVS としての機能範囲 (キーによるアクセスのみ) で利用することも可能

メタデータ

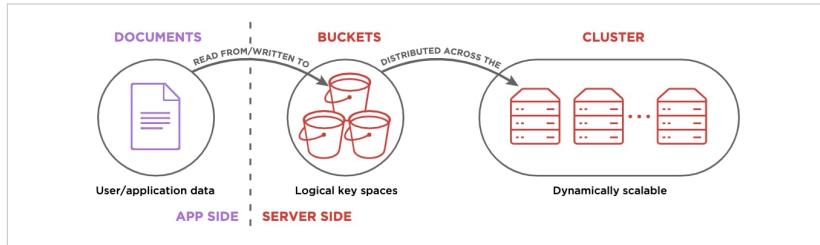
- 固定長
- Couchbase バケットの場合: 56 バイト
- Ephemeral バケットの場合: 72 バイト

5.2 バケット

概要

ドキュメントは、論理的コンテナとしてのバケットに保存されます。
クラスターには、最大 30 のバケットを作成することができます。

^{*2} <https://datatracker.ietf.org/doc/html/rfc8259>



(画像は、Couchbase Under the Hood: An Architectural Overview^{*3}より引用)

種類

バケットには Couchbase バケットと、Ephemeral バケットの 2 種類があります (Couchbase タイプがデフォルト)。

- **Couchbase バケット**は、非同期でデータの永続化を行います (同期的な永続化を指示することも可能)。可能な限り多くのデータをメモリに保持しようとします。
- **Ephemeral バケット**は、データに対する永続化の要件のない場合に用いられ、データはメモリ上にのみ存在します。

■コラム: Memcached バケット

Couchbase Server のバケットの種類には、Couchbase バケット、Ephemeral バケットの他、Memcached バケットがあります。Memcached バケットは、Ephemeral バケットと同じ揮発性データ用のバ

^{*3} https://resources.couchbase.com/c/server-arc-overview?x=V3nd_e

ケットとして、Ephemeral バケット登場以前から存在しています。現在では、Memcached バケットの利用は非推奨 (deprecated) とされています。

5.3 スコープとコレクション

概要

Couchbase Server は、ドキュメント管理のために、スコープとコレクション (Scopes and Collections^{*4}) を提供します。これらを用いて、バケット内でドキュメントを分類し整理することができます。

- コレクションは、ドキュメントをグループ化します。
- スコープは、コレクションをグループ化します。

コレクションの存在意義

ドキュメントをコレクションによってグループ化することによって、ドキュメントの種類（エンティティ）を表現することができます。

コレクションは、リレーションナルデータベースにおけるテーブル相当の位置を占めることになります。

コレクションによって、リレーションナルデータベースから Couchbase Server への移行が容易になります。

標準 SQL を JSON データへのアクセスのために拡張したクエリ言語であ

^{*4} <https://docs.couchbase.com/server/current/learn/data/scopes-and-collections.html>

る N1QL クエリでは、FROM 句にコレクション名を用いますが、RDB からデータ構造を変更せずに、Couchbase Server へそのまま移行した場合、論理的に同じクエリを用いることが可能です。

念のため付け加えると、コレクションによって、リレーションナルデータベースにおけるテーブルのように、ドキュメントの構造が一律に規定される（テーブルスキーマが課せられる）わけではありません。ドキュメント指向データベースとしての柔軟性は損なわれることはありません。

スコープの存在意義

スコープを用いてコレクションをグループ化することによって、エンティティのグループを管理することができます。

スコープは、リレーションナルデータベースにおけるスキーマ相当の位置を占めることになります。

Couchbase Server は、スコープとコレクションによってアクセス権限を管理する機能を提供しています。これによって、マルチテナントが容易になります。

表 5.1: リレーションナルデータベースと Couchbase Server の対比

リレーションナルデータベース	Couchbase Server
サーバー	クラスター
データベース	パケット
スキーマ	スコープ
テーブル	コレクション
レコード (ロウ)	ドキュメント

■コラム: スコープとコレクションの用語法について

「コレクション」という表現は、プログラミングの分野でデータ構造を表す用語として広く浸透しているため、データベースに用いられることに違和感を持つ人もいるかもしれません。ご存知のない方へ向けてお伝えすると、同じドキュメント指向データベースである MongoDB でもドキュメントの格納単位として「コレクション」という用語が用いられています。Couchbase Server は MongoDB に対してその登場自体もドキュメントをグループ化するための機能の追加も後発ですが、いたずらに異なる用語を導入するよりも好ましいと言えるかもしれません。

「スコープ」という表現は、SQL におけるコンテキストとの関係で理解することができます。SQL クエリは、特定のスキーマのコンテキストの元、実行されます。この時、テーブル名が一意に定まるのは、そのコンテキストの「スコープ」においてです（「スコープ」と「コンテキスト」という用語は、プログラミング言語でも、テーブル名のかわりに変数に関して、同じ用法で用いられます）。ちなみに、MongoDB にはスコープに該当する機能が存在しないため、用語の比較 자체が成り立ちません。

機能

スコープとコレクションによるデータ管理によって、以下が可能になります。

- スコープやコレクションごとにアクセス権を割り当てる。
- コレクションにインデックスを付ける（スコープにインデックスを付けることはできません）。

また、Couchbase Server の以下の機能において、スコープとコレクションを利用して、操作の対象を限定することができます。

- N1QL クエリ
- XDCR
- バックアップ
- 統計情報 (cbstats コマンド)

基本仕様

- クラスターごとに最大 1000 個のスコープを作成できます。
- クラスターごとに最大 1000 個のコレクションを作成できます。
- コレクション名は、スコープ内で一意である必要があります。
- アイテム名 (ドキュメントキー) は、コレクション内で一意である必要があります。

作成・管理

スコープとコレクションは、以下を使用して作成・管理できます。

- Web コンソール
- CLI
- REST API
- N1QL
- SDK

デフォルトのスコープとコレクション

すべてのバケットには、デフォルトのスコープが自動的に与えられ、そのスコープにはデフォルトのコレクションがあります。それぞれ、`_default`という名前を持っています。

特定のスコープおよびコレクションを指定しない場合は、`_default`スコープの`_default`コレクションが指定されているものとされます。

また、以下に注意する必要があります。

- デフォルトのスコープは削除できません。
- デフォルトのコレクションは、CLI または REST API のいずれかを使用して削除できます。ただし、一旦削除すると、再作成できなくなります。

スコープとコレクションが初めて導入された 7.0 より前のバージョンの Couchbase Server から 7.0 へアップグレードすると、既存のすべてのドキュメントは、`_default`スコープの`_default`コレクションに自動的に配置されます。これにより、アップグレード時のバージョン間の整合性が維持されます。

ユーザー定義のスコープとコレクション

ユーザー定義のスコープとコレクションに名前を付ける際は、以下の条件を満たす必要があります。

- 長さは 1~30 文字。
- 大文字と小文字が区別されます。
- アルファベット (A-Z, a-z)、と数字 (0-9)、記号として、`_`, `-` と `%` のみを含めることができます (ただし、`_` または `%` で始めることはできません)。

スコープ内の名前空間は、他のスコープの名前空間から独立しています。同一バケット内の複数のスコープに同じコレクション名が存在することができます。同様に、コレクション内の名前空間は、他のコレクションの名前空間から独立しています。その結果、あるバケット内の同じ（または、異なる）スコープの複数のコレクションに、同じドキュメントキーが存在することができます。

5.4 有効期限設定

ドキュメントには有効期限を設定することができます。

■コラム: エディションによる差異

有効期限設定は、エンタープライズエディションでのみ利用可能です。

Time To Live(TTL) をドキュメントに指定することによって、指定された期間の完了後に、そのドキュメントは利用できなくなります（削除されます）。有効期限設定が有効化されている場合、ドキュメントは、TTL で指定された秒数の間存続します。

TTL は、Couchbase バケットと Ephemeral バケットで利用可能です。

バケットおよびコレクションレベルの設定

ドキュメントの有効期限は個々のドキュメント単位で設定するだけではなく、バケットおよびコレクションレベルで設定することが可能です。

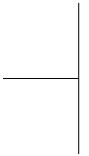
バケットおよびコレクションレベルの設定は、バケット/コレクション内のドキュメントの最大有効期限を決定します。異なるレベルで有効期限が複数設定されている場合、より短い有効期限が採用されます。

バケットおよびコレクションレベルの設定は、設定変更以降に追加された

ドキュメントに対して適用されます。

デフォルト設定

デフォルトでは、有効期限設定は有効化されません。デフォルトの TTL の値は 0 であり、これは TTL が無効であることを示します。



第6章

N1QL クエリ

Couchbase Server の N1QL クエリについて、SQL との差分を中心に解説します (SQL 標準と重複する部分を含めた網羅的な解説であることは意図されていません)。

6.1 基本構造

N1QL は、JSON データと同じ表現を（直接的に、つまり文字列としてエスケープされた形ではなく）含むことができます。

コンストラクションオペレーター

コンストラクションオペレーターと呼ばれる次の記号をクエリ内で直接用いることができます。

{ } : []

ネステッドオペレーター

ネステッドオペレーターと呼ばれる以下の表現を利用することができます。

- **ドットノーテーション** ネストされたオブジェクト（サブドキュメント）へのパスをドット記号（.）を使って表現できます。
- **配列表現** 配列へのアクセスに、インデックス（添字）や、スライス（範囲による指定）を利用することができます。

エスケープ文字

N1QL クエリ中に現れるバケット名などのユーザー定義語句をバッククォート('')を使ってエスケープすることによって、予約語との衝突を回避することができます。

N1QL では、「-」は、予約語に含まれるため、「-」を含むユーザー定義語句を利用する場合、エスケープする必要があります。

クエリコンテキスト

N1QL でスコープとコレクションを活用するために、クエリコンテキストを用いることができます。

これにより、SQL/RDB 経験者にとって、より馴染みやすいクエリ表現を用いることができます。「コンテキスト」という用語自体、SQL/RDB 経験者にとってよく親しまれているものだと思いますが、N1QL でも同じ意味合いで使われています。

まず、コンテキストを使用しないクエリについて見てみます。

```
select * from 'travel-sample'.inventory.airline limit 1;
```

このクエリでは、travel-sample バケットの inventory スコープにある airline コレクションからデータを取得しています。N1QL では、この

のような表現が可能です。

一方、SQL/RDB 経験者が通常、予期するクエリは、以下のようなものでしょう。

```
select * from airline limit 1;
```

クエリコンテキストに、travel-sample バケットの inventory スコープを設定することによって、上記の表現が成立します。

Couchbase Server のクエリ操作用のコマンドラインインターフェースでは、下記のように設定することができます。

```
cbq> \SET -query_context    "travel-sample".inventory;
```

Web コンソールのクエリワークベンチでクエリを実行する場合には、リストボックスからバケット名とスコープ名を選択して、クエリコンテキストを設定することができます。

SDK では、Scope オブジェクトの query メソッドを利用します。また、Bucket オブジェクトの query メソッドを用いて、コンテキストを使用しないクエリの実行も可能です。

Java のサンプルコードを示します。

```
Bucket bucket = cluster.bucket("travel-sample");
Scope scope = bucket.scope("inventory");

QueryResult result = scope.query("select * from airline where country = $country LIMIT 1
    queryOptions().parameters(JsonObject.create().put("country", "France")));
```

以下で用いる N1QL クエリのサンプルでは、クエリコンテキストを指定しない表現を用いています。

6.2 ドキュメントキーの利用

ドキュメントキーの利用は、標準 SQL には見られない、Couchbase Server の N1QL に固有の特徴です。

Couchbase Server では、ドキュメントはドキュメントキーによって一意であるため、利用できる場合には、ドキュメントキーを使うことが最も効果的です。

■コラム：ドキュメントキーが利用できる場合のアプローチ

ドキュメントキーが利用できる場合、必ずしも N1QL クエリを用いて (Query サービスを介して) ドキュメントへアクセスする必要はなく、Data サービスに対して直接リクエストすることで性能を最適化することが可能です。一方、サブクエリを伴うような複雑なクエリの場合などのように、クエリの中でドキュメントキーを効果的に利用できる場面があります。

USE KEYS

下記のように、USE KEYS 句に、ドキュメントキーを指定して、クエリを実行することができます。

```
SELECT *
FROM 'travel-sample'.inventory.airport
USE KEYS "airport_1254";
```

ドキュメントキーは、配列として複数同時に与えることもできます。

```
SELECT *
FROM 'travel-sample'.inventory.airport
USE KEYS ["airport_1254","airport_1255"];
```

USE KEYS のみを利用してクエリを実行する場合、インデックスを作成する必要はありません。

META().id

上記の USE KEYS を利用したクエリは、下記のように META().id を使って書き直すことができます。

```
SELECT *
FROM 'travel-sample'.inventory.airport
WHERE META().id IN ["airport_1254","airport_1255"];
```

META().id を利用してクエリを行う場合、META().id を指定したインデックスが作成されている必要があります。

USE KEYS の構文では、ドキュメントキーの完全一致が必須ですが、META().id を利用する場合、WHERE 句の条件の一部として用いることができます。

例えば、下記のように、LIKE と組み合わせて部分一致検索を行うことができます。

```
SELECT * from MyFirstBucket where Meta().id LIKE "ABC:%";
```

また、検索条件としてのみではなく、次のように、条件に一致するドキュメントキーを検索するために用いることもできます。

```
SELECT META().id FROM ... WHERE ...;
```

このように、META().id は、構文として成立している限り自由に使うことができるため、N1QL 固有の USE KEYS 句よりも自由度が高いと言えます。USE KEYS 句では必要ない、インデックスの作成が必要になるという違いも、ここから自然に理解されるのではないでしょうか。

なお、META() という表現に含まれる括弧は、下記のように、キースペース（または、その別名）を含んだ表現を用いるために存在します。ただし、これまで紹介した例のように、クエリの中でキースペースが一意に決まっている場合には、省略することができます。複数のキースペースが混在しているクエリの中で用いる場合には、キースペースを指定した表現を行います。

```
SELECT *
FROM 'travel-sample'.inventory.airport a
WHERE META(a).id IN ["airport_1254","airport_1255"];
```

6.3 フィールドから値のみを取り出す

RAW | ELEMENT | VALUE

RAW、ELEMENT、VALUE、これらは全て同義語です（以下では、RAW を用います）。

N1QL の実行結果は通常、フィールド名とその値を含む JSON データですが、RAW キーワード用いることで、値のみを取り出すことができます。

比較のため、まず RAW を用いない場合を見てみます。

```
SELECT city
FROM 'travel-sample'.inventory.airport
ORDER BY city LIMIT 5;
```

上記クエリの結果は、以下のように名前と値のペアからなる JSON データの配列になります。

```
[
  {
    "city": "Abbeville"
  },
  {
    "city": "Aberdeen"
```

```
},
{
  "city": "Aberdeen"
},
{
  "city": "Aberdeen"
},
{
  "city": "Abilene"
}
]
```

次に、RAW を用いた場合を見ていきます。

```
SELECT RAW city
FROM 'travel-sample'.inventory.airport
ORDER BY city LIMIT 5;
```

結果は、値のみの配列になります。

```
[
  "Abbeville",
  "Aberdeen",
  "Aberdeen",
  "Aberdeen",
  "Abilene"
]
```

最後に、DISTINCT と組み合わせた場合を見ます。

```
SELECT DISTINCT RAW city
FROM 'travel-sample'.inventory.airport
ORDER BY city LIMIT 5;
```

結果は、値の重複しない配列になります。

```
[  
  "Abbeville",  
  "Aberdeen",  
  "Abilene",  
  "Adak Island",  
  "Addison"  
]
```

6.4 複数の値をまとめて扱う

SQL 経験者にとって、IN 句を使って、複数の値を指定することは、目新しいものではないでしょう。ここでは、N1QL における IN と WITHIN について解説します。

IN

N1QL における IN 句は、SQL の IN 句に近い働きをします。IN 句は、配列のようなフラットなデータシーケンスを対象とします。

```
SELECT * FROM 'travel-sample'.inventory.airline AS t
WHERE country IN ["United Kingdom", "France"];
```

WITHIN

N1QL では、検索条件に複数の値を指定する際に、WITHIN 句を使って、JSON データのネストした構造を対象とすることができます。

JSON データに対する条件指定において、IN は、トップレベルの要素を対象とするのに対して、WITHIN は、その子、および子孫を含めて対象とします。

具体的に見ていきます。例えば、以下のクエリでは、WITHIN 句に (別名を介して)hotel コレクションを指定しています。

```
SELECT * FROM 'travel-sample'.inventory.hotel AS t WHERE "Walton Wolf" WITHIN t;
```

以下は結果の例です。

```
[  
  {  
    "t": {  
      "id": 10851,  
      "name": "Tantallon House B&B",  
      "address": "Gilsland, CA8 7DA",  
      "country": "United Kingdom",  
      "reviews": [  
        {  
          "author": "Walton Wolf",  
          "content": "Myself and a mate stayed here the last week of April 2010. ...",  
          "date": "2014-06-07 03:54:50 +0300",  
          "ratings": {  
            "avg": 5.0,  
            "count": 1  
          }  
        }  
      ]  
    }  
  }  
]
```

```
        "Cleanliness": 4,
        "Location": 4,
        "Overall": 4,
        "Rooms": 2,
        "Service": 3,
        "Sleep Quality": 4,
        "Value": 3
    }
},
{
    "author": "Catharine Funk",
    "content": "My first trip to new york city was a disaster because of this hotel",
    "date": "2012-01-02 21:29:27 +0300",
    "ratings": [
        ...
    ],
    {
        ...
    },
    ...
],
...
}
]
```

検索結果として、1つのドキュメントが該当しています。トップレベルの reviews フィールドは、値を配列として持っております、その要素はオブジェクトです。配列の先頭要素のオブジェクトの author フィールドの値が、検索条件に用いられた "Walton Wolf" であることが分かります。

6.5 配列へのアクセス

N1QL で、配列へのアクセスに用いられる構文は、ARRAY または FIRST から始まり、END で終わります。

下記に基本構文を示します(簡単な構文からはじめ、徐々に複雑な構文を紹介していきます)。

この構文を利用して、配列の要素に対して条件指定を行って要素を抽出したり、配列が要素として値ではなくオブジェクトを持っている時、そのオブジェクトのフィールドを配列として取り出すことができます。

```
( ARRAY | FIRST ) var1 FOR var1 ( IN | WITHIN ) expr1 END
```

ARRAY

配列の要素全体を扱います(プログラミング言語の文脈で言うところの、シーケンスに対する map 操作にあたります)。

FIRST

ARRAY を FIRST に置き換えることで、配列のはじめの要素のみを扱うことになります。

WHEN

下記構文のように、WHEN 句を使って、条件指定を付け加えることが可能です(配列の要素に対して filter 操作を実行します)。

```
( ARRAY | FIRST ) var1 FOR var1 ( IN | WITHIN ) expr1 [ ( WHEN cond1 [ AND cond2 ] ) ] END
```

以下にクエリとその結果のサンプルを示します(構文の理解を容易にする

第6章 N1QL クエリ

6.5 配列へのアクセス

ため、クエリ内でデータを定義しています)。

```
select ARRAY v FOR v IN [1, 2, 3, 4, 5] WHEN v > 2 END as res;
```

```
"results": [
  {
    "res": [
      3,
      4,
      5
    ]
  }
]
```

下記のように、配列要素がオブジェクトの場合、フィールドに対して条件指定を行ったり、特定のフィールドを取り出すことができます。

```
SELECT ARRAY v.flight FOR v IN schedule WHEN v.utc > "19:00" AND v.day = 5 END AS fri_evening
FROM 'travel-sample'.inventory.route
LIMIT 5
```

最後に、これまで説明した構文を踏まえ、配列を複数 (var1、var2) 指定した、下記のような表現を用いることができます。

```
( ARRAY | FIRST ) var1 FOR var1 ( IN | WITHIN ) expr1 [ ,var2 ( IN | WITHIN ) expr2 ]*
[ ( WHEN cond1 [ AND cond2 ] ) ] END
```

6.6 配列を検索条件に利用

配列を検索条件の中で用いる際の構文は、ANY または EVERY から始まり、END で終わります。SATISFIES で条件を指定します。

```
( ANY | EVERY ) var1 ( IN | WITHIN ) expr1  
[ , var2 ( IN | WITHIN ) expr2 ]*  
SATISFIES condition END
```

条件が真となった際に、その配列を持つドキュメントが、検索結果に含まれます。

ANY

配列に条件で指定した要素が1つでも含まれる場合、真となります。
下記に利用例を示します。

```
SELECT *  
FROM retail.east.order o  
WHERE ANY item IN o.lineItems SATISFIES  
item.count >= 5 END
```

EVERY

配列の全ての要素が、指定した条件に一致する場合、真となります。

下記は、上記例のクエリの ANY を EVERY に変更しただけですが、条件に一致するドキュメントの範囲が狭まる（可能性が高い）ことは理解しやすい

と思います。

```
SELECT *
FROM retail.east.order o
WHERE EVERY item IN o.lineItems SATISFIES
item.count >= 5 END
```

6.7 サブドキュメントの階層操作

NEST

NEST は、外部のドキュメントを、子ドキュメントとして親ドキュメントの中に(サブドキュメントとして)埋め込みます。結合の特別なタイプであると言った方が、理解しやすいかもしれません(ここで登場する、ON KEYS 句については、NEST を使わない結合でも利用します)。

具体的な利用法として、関連のあるデータをドキュメントキーを介した参照関係を用いて、複数のドキュメントとしてモデル化した際に活用する事ができます。それらのドキュメントを結合するために、ON KEYS を用いて、結合対象とするドキュメントのキーを指定します。

以下の2種類のデータに対するクエリについて考えてみます。リレーションモデルの解説では、よく知られた例である「受注伝票」と「受注明細」データが表現されています。

まず、「受注伝票」を表すデータ (retail.east.order コレクション) のサンプルを示します。

```
{
  "order_id":1234,
```

```
"customer_id": "34567",
"total_price": "65.5",
"lineitems": ["o11", "o12"]
}
```

次に、「受注明細」を表すデータ (retail.east.lineItem コレクション) のサンプルを示します。ドキュメントキーは、lineitem_id の値と一致していると考えてください。

```
{
  "lineitem_id": "o11",
  "item_id": "789",
  "qty": "3",
  "itemprice": "5.99",
  "base_price": "17.97",
  "tax": "0.75",
  "total_price": "18.22"
}
```

```
{
  "lineitem_id": "o12",
  "item_id": "234",
  "qty": "5",
  "itemprice": "10.00",
  "base_price": "50.00",
  "tax": "0.75",
  "total_price": "50.75"
}
```

下記に、これらのデータを結合するクエリを示します。

```
SELECT ordr.order_id,
ARRAY {"item_id": l.item_id, "quantity":l.qty} FOR l IN line END as items
FROM retailsample.east.order ordr
NEST retailsample.east.lineItem line
ON KEYS ordr.lineitems
```

結果は、以下のようになります。

```
[
{
  "items": [
    {"item_id": "789", "qty": "3"},
    {"item_id": "234", "qty": "5"}
  ],
  "order_id": "1234"
}]
```

UNNEST

UNNEST は、ネストされたオブジェクト（サブドキュメント）をドキュメントのトップレベルに表示するために用います。

下記のようなネストされた構造を持つデータを想定します。

先ほどとは逆に、受注伝票データが受注明細データを含む構造になっています。

```
{
  "ordId": "ORDER-0001",
```

```
"status": "Shipped",
"items": [
  {
    "prodId": "AAA-222",
    "qty": 1
  },
  {
    "prodId": "BBB-333",
    "qty": 2
  },
  {
    "prodId": "CCC-444",
    "qty": 3
  }
]
```

UNNEST は、このようなネストされた（サブドキュメントを含む）ドキュメントへのクエリの結果を下記のようなテーブル構造として利用したい時に使われます。

ordId	Status	prodId	qty
ORDER-0001	Shipped	AAA-222	1
ORDER-0001	Shipped	BBB-333	2
ORDER-0001	Shipped	CCC-444	3

下記のように、サブドキュメントに対して UNNEST を指定し、as で別名をつけたものを SELECT 句の中で使用します。

```
SELECT ord.ordId, ord.status, item.* FROM retail.east.order ord UNNEST items as item
```

これにより、下記のようなフラットな構造の JSON が取り出されます。

```
[  
  { "ordId": "ORDER-0001", "status": "Shipped", "prodId": "AAA-222"  
   , "qty" : 1 },  
  { "ordId": "ORDER-0001", "status": "Shipped", "prodId": "BBB-333"  
   , "qty" : 2 },  
  { "ordId": "ORDER-0001", "status": "Shipped", "prodId": "CCC-444"  
   , "qty" : 3 }  
]
```

なお、上記のデータは、先述のテーブル形式データとの比較のために整形しています。実際には、フィールドの出現順はこの通りではないことにご注意ください。JSON オブジェクトのフィールドはその登場順序に意味を持ちません。

6.8 データ型

ここでは、Couchbase Server の N1QL 固有のデータ型 (Data Types^{*1})について理解するために、標準 SQL に含まれる関連性の高いデータ型とあわせて解説します。

SQL 利用者にとって、「NULL」という予約語や、IS [NOT] NULL という比較演算は、馴染みのあるものでしょう。NULL は、データが存在しないことである、と言ったりします。より正確には、値が存在しない状態を表しています。ここで、データと値の違いはどこにあり、なぜ後者がより正確なのでしょうか？「データ」という表現は曖昧に取れる一方、「値」という言葉には、数学でいうところの X の値という使い方に見られるように（もちろん、プログラミングでいうところの、変数の値、でもいいわけですが）、単にそれ自体で存在しているデータを超えた含意があります。リレーションナルデー

^{*1} <https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/datatypes.html#datatypes>

タベースでは、カラムは一定のデータ型を用いて定義されますが、数値型であれば 0、文字列型であれば空文字（長さ 0 の文字）のようなデータ型固有の表現とは別に、値そのものが存在していない状態が NULL で表現されます。JSONにおいても、「フィールドの値が未定義」であることを意味する「null」という予約語が定義されています。一方、JSON のように、データ自体にデータ構造の情報が含まれる（データとは別にスキーマが存在していない）、データにあっては、「フィールドが未定義」のケースを考慮する必要があります。

JSON データに対して、N1QL によるクエリを実行した場合、他のドキュメントには存在するフィールドが、一部のドキュメントには存在しないことがあります（RDB/SQL では、テーブル内のすべてのレコードが同一のスキーマに従うため、このような状況が起こりません）。N1QL には、そのようなギャップを埋めるため、MISSING というキーワード（データ型）が用意されています。

以下、N1QL におけるデータ型としての、NULL と MISSING の定義を確認した後に、N1QL における、存在しない値に対する比較演算について見ていきます。

MISSING

N1QLにおいて MISSING は、JSON ドキュメントの中にそのフィールド（名前と値のペア）が存在していない（欠落している）ことを表します。

N1QL では、SELECT 句で指定されたフィールドが存在しない（MISSING である）データがある場合、（内部的には MISSING リテラル式で処理されます）検索結果データでは、（JSON データにおける未定義の値である） null に変換されます。

NULL

N1QL は、キーワード NULL を使用して空の値を表します。データ挿入、更新時に、フィールドの値を NULL にすることができます。

また、NULL 値は、ゼロ除算や間違ったタイプの引数の受け渡しなど、特定の操作によっても生成され得ます。

なお、N1QL の NULL では大文字と小文字が区別されません。たとえば、`null`、`NULL`、`Null`、および `nUll` はすべて同等です。

6.9 比較演算

ここでは、Couchbase Server の N1QL 固有の比較演算子 (Comparison Operators^{*2}) について見ていきます。

`IS [NOT] NULL` । `MISSING` । `VALUED` 演算子を使用すると、データセット内の値や属性の存在（または不在）に基づいて条件を指定できます。

`IS [NOT] NULL` と `IS [NOT] MISSING` は、データ型の定義から想像される通りの挙動となります。具体的なクエリと結果の例を見るのが、理解に役立つでしょう。

IS NULL

```
SELECT fname, children
FROM tutorial.sample.person
WHERE children IS NULL
```

^{*2} <https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/comparisonops.html>

```
{  
  "results": [  
    {  
      "children": null,  
      "fname": "Fred"  
    }  
  ]  
}
```

IS MISSING

```
SELECT fname  
FROM tutorial.sample.person  
WHERE children IS MISSING
```

```
[  
  {  
    "res": [  
      3,  
      4,  
      5  
    ]  
  }  
]
```

IS VALUED

NULL と MISSING は、JSON データ上明らかに異なっており、区別する方法が必要である一方、アプリケーションにおける検索条件としてはこれらを

区別する必要がない場合が考えられます。その際に AND や OR を使って複数の検索条件を並置する代わりに、N1QL では、IS [NOT] VALUED という比較演算が利用可能です。

- IS VALUED は、値が MISSING でも NULL でもない場合、真になります。
- IS NOT VALUED は、値が MISSING あるいは NULL である場合、真になります。

6.10 結合 (JOIN)

N1QL で利用することのできる、以下の 3 つの結合 (JOIN^{*3}) のタイプについて解説します。

- ANSI JOIN
- ルックアップ (Lookup) JOIN
- インデックス (Index) JOIN

ANSI JOIN 以外の 2 つの種類の JOIN では、N1QL 独自のキーワードが用いられます。それらは、ON KEYS と ON KEY ... FOR です。これらは、JSON ドキュメントのドキュメントキーによる結合をサポートするために用意されています。

ここでは、以下のデータモデルを用いて、例示による整理を行います。

- `airline`(航空会社) と `route`(経路) という、2 つの種類のデータモデルがある
- `route` データモデルは、`airline`への参照を含む。参照の値は `airline` ドキュメントのキーと一致している。

^{*3} <https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/join.html>

ANSI JOIN

ANSI JOIN は、N1QL 独自のキーワードを用いない、SQL の一般的な JOIN です。ANSI JOIN でドキュメントキーを用いた結合を行うために、`META().id` を用いることができます。この場合、`META().id` を含む適切なインデックスが作成されている必要があります。

以下に、基本構文を示します。

```
lhs-expr
JOIN rhs-keyspace
ON any join condition
```

下記は、`META().id` を用いた ANSI JOIN の例です。

```
SELECT *
FROM 'travel-sample'.inventory.route r
JOIN 'travel-sample'.inventory.airline a
ON r.airlineid = META(a).id
```

■コラム: N1QL の JOIN と標準 SQL との違い

N1QL の ANSI JOIN には、SQL 標準と比べて、以下のようないくつかの制約があります。

- FULL [OUTER] JOIN はサポートされていません。
- CROSS JOIN はサポートされていません。
- RIGHT [OUTER] JOIN は、JOIN 連結の最初（または、それのみ）

である必要があります。

ルックアップ JOIN

ルックアップ JOIN では、ON KEYS 句を用います (複数形の KEYS です)。ドキュメントキーを用いた結合を行うために、インデックスを利用する必要はありません。

以下に、基本構文を示します。

```
lhs-expr  
JOIN rhs-keyspace  
ON KEYS lhs-expr.foreign-key
```

上述の ANSI JOIN の例と、同内容のクエリを下記のように表現することができます。

```
SELECT *  
FROM 'travel-sample'.inventory.route r  
JOIN 'travel-sample'.inventory.airline  
ON KEYS r.airlineid
```

インデックス JOIN

ON KEY ... FOR 句を用います (単数形の KEY です)。インデックス

JOIN では、適切なインデックスが作成されている必要があります。

以下に、基本構文を示します。

```
lhs-keyspace
JOIN rhs-keyspace
ON KEY rhs-keyspace.idx_key
FOR lhs-keyspace
```

下記のクエリ例では、ルックアップ JOIN のケースと FROM ... JOIN ... で指定されているキースペース（コレクション）の順序が異なっていることに注目してください。

```
SELECT *
FROM 'travel-sample'.inventory.airline a
JOIN 'travel-sample'.inventory.route r
ON KEY r.airlineid
FOR a
```

上記のクエリを実行するためには、下記のようなインデックスが必要です。

```
CREATE INDEX route_airlineid ON 'travel-sample'.inventory.route(airlineid);
```

インデックス JOIN を使用すると、結合句の方向を反転できます。キーによる結合以外の条件指定のため、ルックアップ JOIN が利用できない場合、インデックス JOIN が使用できます。

サンプルを示した解説は、データモデルの説明を含めて、長々としたもの

になってしまふため、ドキュメント^{*4}の記述に譲ります。

6.11 ユーザー定義関数

Couchbase Server の N1QL では、ユーザーが関数を定義して、組み込み関数と同じように、任意の式で呼び出すことができます。

■コラム: エディションによる差異

ユーザー定義関数は、エンタープライズエディションでのみ利用することができます。

ここでは、ユーザー定義関数を用いて、何が実現できるかを把握するための情報を提供します。実際に利用するために必要となる、定義方法や呼び出し方を含めた、具体的な詳細には触れませんが、必要に応じて、ドキュメント^{*5}を参照してください。

ユーザー定義関数の種類

ユーザー定義関数は CREATE FUNCTION^{*6}ステートメントを用いて定義します。

定義の仕方の異なる次の 2 つの種類のユーザー定義関数があります。

- ・ インライン関数は、N1QL クエリを使用して定義されます。複雑な式

^{*4} <https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/join.html#examples-4>

^{*5} <https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/userfun.html>

^{*6} <https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/createfunction.html>

や、何度も用いられる式に名前を付けて定義し、再利用することができます。

- 外部関数は、N1QL ではなく、プログラミング言語を使用して定義されます。この機能により、N1QL 式を使用して定義するのが困難または不可能な関数を作成することができます。構文上は、他の言語の利用も想定される書式が用いられていますが、現時点ではサポートされているプログラミング言語は JavaScript のみです。

関数呼び出しの種類

ユーザー定義関数には次の 2 つの呼び出し方法があります。

- N1QL の任意の式中で、組み込み関数と同じように、呼び出すことができます。ただし、Couchbase Server のユーザー定義関数は、単純なインライン関数を超えた高い自由度を持ち、関数の中でドキュメントの変更などを行うことができる反面、このような「副作用」を持つ、ユーザー定義関数を N1QL 式の中から呼び出した場合には、エラーが発生することに注意が必要です。
- EXECUTE FUNCTION^{*7}ステートメントを使って、ユーザー定義関数を単体で直接実行することが可能です。これは、ユーザー定義関数をテストする際に用いることができる他、EXECUTE FUNCTION ステートメントでは副作用のある関数を実行することが可能なため、RDBMS におけるストアドプロシージャのような使い方ができます（なお、Couchbase Server では、RDBMS におけるトリガに類似する機能として Eventing サービスを利用することができます）。

^{*7} <https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/execfunction.html>

■コラム: エディションによる差異

Eventing サービスは、エンタープライズエディションでのみ利用することができます。

■コラム: エディションによる差異

クエリに関連した、コミュニティエディションには含まれない、エンタープライズエディションの機能として、以下があります。

- ウィンドウ関数
- FLEX インデックス（クエリからの全文検索インデックス利用）
- コストベースオプティマイザ（CBO）
- 無制限のクエリ同時実行
- N1QL アグリゲートプッシュダウン
- N1QL リクエストの監査
- クエリのモニタリング

第7章

インデックス

Couchbase Server における、インデックス機能を RDB との比較を交えながら説明します。

Couchbase Server では、RDB とは異なり、クエリの実行に際して、基本的に、対応するインデックスの存在が前提となります。

7.1 概要

用語法に関する注釈

Couchbase Server におけるインデックスは、グローバルセカンダリインデックス (GSI) と呼ばれています。別に「プライマリインデックス」、「セカンダリインデックス」という表現も用いられますが、これらはいずれも、グローバルセカンダリインデックス (GSI) の中のカテゴリーとなります。(以下は、「プライマリインデックス」のメタデータの例です。"is_primary": trueかつ"using": "gsi"であることが分かります)

```
{
```

```
"datastore_id": "http://127.0.0.1:8091",
"id": "1c23ecbf1f7cd99e",
"index_key": [],
"is_primary": true,
"keyspace_id": "travel-sample",
"name": "#primary",
"namespace_id": "default",
"state": "online",
"using": "gsi"
}
```

来歴

グローバルセカンダリインデックス (GSI) が登場する前には、データの検索のために、MapReduce ビュー (View) という機能が使われていました（現在も互換性維持のために残されてはいますが、将来のリリースでは廃止されるとされています）。この View が、データが存在するノードに対して「ローカル」であったことに対して、ローカリティに依存しない、クラスター単位で管理されるインデックスという意味で、「グローバル」という表現が用いられていると解釈することができます。このように、「グローバル」という語句は、別のローカルなインデックスとの対比で用いられているのではなく、Couchbase Server には、ローカルセカンダリインデックス (Local Secondary Index) という概念は存在しません。

また、「セカンダリ」という形容は、RDB におけるセカンダリインデックスのように任意の数のインデックスを定義できる、という意味合いで受け取ることができます。

このように後続のテクノロジーであるという経緯から、Couchbase Server の過去のバージョンでは GSI としてインデックスを作成する際に明示的に USING GSI という句が用いられていました。6.5 以降のバージョンでは、 USING GSI の利用はオプションであり、省略可能です。

RDB におけるインデックスとの違い

Couchbase Server のインデックスには、RDB におけるインデックスと異なる、以下のような特徴があります。

- そのクエリを実行するために必要なインデックスが存在しない場合、処理が失敗します。
- インデックスは、非同期で維持されます。
- 同じ内容のインデックスを別名で複数定義できます。

これらの特徴については、それぞれ関連する章で、背景やその特徴を踏まえた利用方法を説明していきます。

ストレージモデル

Couchbase Server のグローバルセカンダリインデックス (GSI) には以下の 2 つのストレージモデルがあります。

- スタンダード GSI
- メモリ最適化 (Memory-optimized) GSI

■コラム: エディションによる差異

メモリ最適化 GSI は、コミュニティエディションでは利用することができません。

スタンダード GSI は、ForestDB ストレージエンジンを使用して B ツリーインデックスを格納しています。また、最適なワーキングセットのデータをバッファに保持します。

メモリ最適化インデックスは、すべてのインデックスデータをメモリに持ち、ロックフリースキップリストを使用してインデックスを保持しています。メモリ最適化インデックスは、インデックス更新時、より高速に処理が行われます。

スタンダード GSI、メモリ最適化 GSI のいずれを使うかは、クラスター単位の設定となり、クラスターの初期構成時に、そのクラスターで使用するインデックスのタイプを選択することになります。

インデックス情報の確認

`system:indexes` に対してクエリすることにより、インデックスに関する情報を確認することができます。

```
SELECT * FROM system:indexes WHERE name = '#primary';
```

```
[  
  {  
    "indexes": {  
      "bucket_id": "travel-sample",  
      "datastore_id": "http://127.0.0.1:8091",  
      "id": "804d33907ce31e27",  
      "index_key": [],  
      "is_primary": true,  
      "keyspace_id": "airline",  
      "name": "#primary",  
      "namespace_id": "default",  
      "scope_id": "inventory",  
      "state": "online",  
      "using": "gsi"  
    }  
  }
```

```
}
```

上記例のように、インデックスが存在するノード (`datastore_id`) や状態 (`state`) など、インデックスに関する情報を確認することができます。

■コラム: エディションによる差異

Index サービスにおける、コミュニティエディションには含まれない、エンタープライズエディションの機能として、以下があります。

- インデックスのパーティション化
- インデックスレプリカ
- メモリ最適化インデックスストレージ
- Plasma (インデックス用の高速ストレージエンジン)
- インデックスアドバイザー

7.2 プライマリインデックス

プライマリインデックスは、キースペース単位で指定するインデックスです。

■コラム: ドキュメントの一意性について

Couchbase Server では、プライマリインデックスとは関係なく、ドキュメントキーはキースペースにおいて一意です。

概要

クエリ実行時に、プライマリインデックスが使用されるケースとしては、以下があります。

- クエリにフィルタ（述語・WHERE 句）がない場合
- クエリにフィルタ（述語・WHERE 句）があるが、該当するセカンダリインデックスがない場合

Couchbase Server では、クエリを実行するために必要なインデックスが存在しない場合、処理が失敗します。プライマリインデックスは、ドキュメントの特定のフィールドに限られない、キースペース（下記例の場合、airline コレクション）全体に対して、そのような制約を回避するための最も簡便な方法になります。プライマリインデックスが存在する場合、クエリの検索条件に使われている項目を個別に指定してインデックスを作成しなくても、クエリを実行することが可能になります。ただし、結合を行う際など、プライマリインデックスで代用できない場合があります。

定義方法

```
CREATE PRIMARY INDEX ON 'travel-sample'.inventory.airline;
```

作成されたインデックスの定義を、Web コンソール等で、確認すると分かりますが、上記の DDL(データ定義言語) は、下記の DDL を実行する際の糖衣構文 (syntax sugar) にあたるものです。

```
CREATE PRIMARY INDEX '#primary' ON 'travel-sample'.inventory.airline
```

Couchbase Server のインデックスには名前を定義するのが通常であり、上掲の省略形は、プライマリインデックスに特有のものです。次の例に示すように、明示的に、名前を付けて定義する構文が基本となります。

```
CREATE PRIMARY INDEX <インデックス名> ON <キースペース名>;
```

■コラム：インデックス設計のベストプラクティス

プライマリインデックスに頼った検索は、検索条件の内容に関わらず、キースペース（上記例の場合、`airline` コレクション）内のドキュメントに対する全件スキャンを実行することに留意が必要です。これは、ドキュメントキーを検索条件（WHERE 句）に用いたクエリの場合も同様です（ドキュメントキーを用いたセカンダリインデックスの定義については別に触れます）。

検索条件のないクエリ、つまりコレクション（エンティティ）から全てのデータを取得するクエリを実行する場合には、プライマリインデックスを利用する事になりますが、その場合も、条件を指定した個別の検索要件のためには、別にセカンダリインデックスを作成することによって、不要な全件スキャンが発生しないようにすることが重要です。

■コラム: アドホッククエリ要件について

ここで説明したインデックスの仕様について、柔軟性に欠けると思われる方もいるかもしれません。Couchbase Server のコア機能は、アプリケーションのバックエンドデータベースとして、設計・最適化されており、このインデックスの仕様は、一般的なアプリケーションでは、ユーザーにより恣意的に記述されるクエリ（アドホッククエリ）は発生しない（必要とされない）ことを反映しているといえるでしょう。柔軟性は性能とのトレードオフになり得ることを考えると、これは合理的な仕様だといえます。

一方、コア機能を補完する別のサービスが提供されており、例えば、データ分析者によるヒストリカルデータへのアドホックなクエリという要件については、（エンタープライズエディションで提供される）Analytics サービスが対応します。また、自由な語句の入力による全文検索という要件については、（本書では扱わない）Search サービスが提供されています。

7.3 セカンダリインデックス

概要

JSON ドキュメントの特定の要素（ドキュメントキー指定のためのメタデータ利用を含む）に対して設定するインデックスは、セカンダリインデックスと呼ばれます。

定義方法

まずは以下、JSON のフィールドの扱いの違いによる定義方法を見ていきまます。

単純にフィールド名を指定した場合、ドキュメントのトップレベルのフィールドを指します。

```
CREATE INDEX travel_name ON 'travel-sample'.inventory.airline(name);
```

このキースペースにおいて、`name` は次のような単純なスカラー値です。

```
{ "name": "Air France" }
```

スカラー値のみではなく、JSON の他のデータ形式（配列、オブジェクト）を値として持つフィールドを用いることも可能ですが、この場合、述語条件の右側の値は同様に構造化されたデータ形式（配列、オブジェクト）である必要があります。

以下のように、ドット（.）表記（ノーテーション）により、ネストされたオブジェクトのフィールドを指定することもできます。

```
CREATE INDEX travel_geo_alt on 'travel-sample'.inventory.landmark(geo.alt);
CREATE INDEX travel_geo_lat on 'travel-sample'.inventory.landmark(geo.lat);
```

ここで、`geo` は、次のようなドキュメント内に埋め込まれたオブジェクト

です。

```
{  
  ...,  
  "geo": {  
    "alt": 12,  
    "lat": 50.962097,  
    "lon": 1.954764  
  }  
}
```

ドキュメントキーの利用

セカンダリインデックスのキーとして、ドキュメントキーを用いることができます。ドキュメントキーは、ドキュメント内の情報ではないため、META() キーワードを用います。

```
CREATE INDEX travel_info ON 'travel-sample'.inventory.airline(META().id);
```

7.4 配列インデックス

Couchbase Server のインデックス定義 DDL は、配列に対してインデックスを作成するための、特別な構文を備えています。

要素オブジェクトの特定フィールド指定

JSON ドキュメントは、スカラー、オブジェクト（サブドキュメント）、またはそれらの配列をフィールドとして持つことができます。

具体的に理解するため、次の配列について考えてみます。

```
"schedule": [
  {
    "day": 0,
    "flight": "AF198",
    "utc": "10:13:00"
  },
  {
    "day": 0,
    "flight": "AF547",
    "utc": "19:14:00"
  },
  {
    "day": 0,
    "flight": "AF943",
    "utc": "01:31:00"
  },
  {
    "day": 1,
    "flight": "AF356",
    "utc": "12:40:00"
  },
  {
    "day": 1,
    "flight": "AF480",
    "utc": "08:58:00"
  },
  {
    "day": 1,
    "flight": "AF250",
    "utc": "12:59:00"
  }
]
```

```
}
```

この `schedule` 配列に見られるような構造に対して、配列の要素であるオブジェクトの特定のフィールドにインデックスを付ける方法を次に示します。

```
CREATE INDEX idx_flight
ON 'travel-sample'.inventory.route
( DISTINCT ARRAY v.flight FOR v IN schedule END );
```

上のインデックス定義 DDL は、以下の基本構文を元にしています。

```
ARRAY <インデックスキー> FOR v IN <配列名> END
```

`v` は、`schedule` 配列内の各要素オブジェクトを参照するために宣言された変数です。`v.flight` は、配列の要素オブジェクトの `flight` フィールドを示しています。

上記の例では、`DISTINCT` キーワードにより、インデックスからキーの重複が除かれます。全てのキーを格納する場合は `ALL` 修飾子を利用します。

上記のインデックスにより、下記のように `schedule.flight` に対して検索を行うことが可能になります。

```
SELECT * FROM 'travel-sample'.inventory.route
WHERE ANY v IN schedule SATISFIES v.flight LIKE 'UA%' END;
```

要素オブジェクトの複数フィールド指定

次に、schedule(フライトスケジュール) の flight(フライト番号) と day(運行曜日) の情報を使ってインデックスを作成します。

複数のフィールドを指定する場合は、 [...] を使います。

```
CREATE INDEX idx_flight_and_day ON 'travel-sample'.inventory.route  
  (DISTINCT ARRAY [v.flight, v.day] FOR v IN schedule END);
```

次のクエリは、週の2番目の曜日にスケジュールされたフライト番号「US681」のルートを検索します。

```
SELECT META().id FROM 'travel-sample'.inventory.route  
 WHERE ANY v in schedule SATISFIES [v.flight, v.day] = ["US681", 2] END;
```

特定条件で選択された配列要素を利用

次に、週の最初の3日間に運行しているサンフランシスコからのフライトを指定して、インデックスを作成します。

下記のDDLでは、「ARRAY ... END」構文の内部で、「WHEN v.day < 4」という検索条件が用いられているのが分かります。

```
CREATE INDEX idx_flight_early_week
ON 'travel-sample'.inventory.route
( ALL ARRAY v.flight FOR v IN schedule WHEN v.day < 4 END )
WHERE sourceairport = "SFO";
```

週の初日にスケジュールされた「UA」のフライトを検索するクエリを示します。

```
SELECT * FROM 'travel-sample'.inventory.route
WHERE sourceairport = "SFO" ①
AND ANY v IN schedule SATISFIES (v.flight LIKE 'UA%') ②
AND (v.day = 1) END; ③
```

インデックス `idx_flight_early_week` は、次のような関係において、上のクエリの対象となります。

- ① 述語中の「`sourceairport = "SFO"`」の箇所が、インデックスの `WHERE` 句に一致します。
- ② 述語中、`ANY-SATISFIES` 句の `schedule` 配列に関する検索のキーとして、`v.flight` が一致します。
- ③ `ANY-SATISFIES` 句のもう 1 つの条件「`v.day = 1`」は、インデックス定義の `WHEN` 句の条件「`v.day < 4`」に含まれます。

要素オブジェクトの全てのフィールドを利用

次の DDL は、`schedule` 配列のすべての要素を含むインデックスを作成します。

```
CREATE INDEX idx_sched_simple  
ON 'travel-sample'.inventory.route(ALL schedule);
```

次のクエリは、これまでよりも詳細な条件が指定された特定のスケジュールに一致するルートを検索します。

```
SELECT * FROM 'travel-sample'.inventory.route  
WHERE ANY v IN schedule  
SATISFIES v = {"day":2, "flight": "US681", "utc": "19:20:00"} END;
```

7.5 インデックス論理設計

インデックスの論理設計においては、要件に対して複数の選択肢があり得ます。クエリの性能を最適化するために適切な選択を行うことが重要です。

複合 (Composit) インデックス

複数のキーを持つインデックスを定義することが可能です。

```
CREATE INDEX travel_info ON 'travel-sample'.inventory.airline(name, id, icao, iata);
```

複合 (Composit) インデックスには、**先行属性 (leading attributes)** マッチという性質があります。

上記インデックスは、`name` と `id` を用いた検索には利用可能ですが、`name` を含まない、その他のフィールドを使った検索では、このインデックスは

使用されません。つまり、上記のインデックスでは4つのフィールドが指定されていますが、可能な組み合わせは、「name のみ」、「name と id」、「name と id と icao」、「name と id と icao と iata」の4通りになります。

複合インデックスの先行属性マッチという性質から、検索条件の組み合わせの数にあわせて、必要なインデックスの数が増えてしまう、という煩雑さが生じます。そのような状況を回避するための機能として、適応(Adaptive)インデックスが用意されています。

カバリング(Covering)インデックス

クエリがインデックスのキーのみを参照している場合(つまり、検索条件としてだけでなく、SELECT句の中でもインデックスのキーのみが指定されている場合)、クエリエンジンは、データノードからフェッチすることなく、インデックススキャン結果を利用してクエリに応答することができます。

検索条件に用いる必要のないフィールドをあえてインデックスキーに含めて、複合(コンポジット)インデックスを作成することで、クエリの性能を向上することができる場合があります。

一方、インデックスの維持に必要なリソース負担が増加することに注意が必要です。

部分(Partial)インデックス

インデックス定義DDLの中で、WHERE句を用いることで、本来のデータに対して、部分的にインデックスを作成することができます。

下記の例では、activityフィールドの値としてeatを持つドキュメントに対してのみインデックスが作成されます。

```
CREATE INDEX travel_eat ON 'travel-sample'.inventory.landmark(name, id, address)
WHERE activity='eat';
```

このインデックスがクエリに利用されるためには、そのクエリにも同じ条件が含まれている必要があります。

部分インデックスを使用する目的として以下が考えられます。

- 必要な検索要件のみにインデックス化するデータの範囲を限定することによって、検索性能とリソース消費を最適化する。
- (性別などのように) 選択肢が限定的な(カーディナリティが低い)データを検索する場合に、選択肢毎に複数のインデックスに分割することによって、(例えば、分散環境の効果と組み合わせて) 性能を最適化する。

関数 (Functional) インデックス

関数を適用したフィールドをキーとして使用しインデックスを作成することができます。

インデックスには、関数適用後の値が格納されます。

複数ドキュメントに跨がる集計関数 (SUM、AVG、COUNT 等) は使用できません。

```
CREATE INDEX travel_cxname ON 'travel-sample'.inventory.airport(LOWER(name));
```

上の例は、name フィールドの値として、大文字と小文字が混在しているケースを想定して、LOWER 関数を使って値を小文字に統一しています。

このインデックスがクエリに利用されるには、下のように、条件式の左辺、関数 LOWER と関数へのパラメータ name の組み合わせがインデックスの定義と一致している必要があります。

```
SELECT * FROM 'travel-sample'.inventory.airport  
WHERE LOWER(name) = "villeneuve-sur-lot";
```

より複雑な式の組み合わせを使用することもできます。例えば以下のように、計算式を指定することも可能です。

```
CREATE INDEX travel_cx1 ON 'travel-sample'.inventory.airport  
(LOWER(name), ROUND(geo.alt * 0.3048));
```

適応 (Adaptive) インデックス

DISTINCT PAIRS 構文を利用して、ドキュメントの複数のフィールドに対して、インデックスを付けることができます。この場合、複合インデックスの場合と異なり、フィールドの指定順序にかかわらず、いずれかのフィールド (の組み合わせ) を検索条件としたクエリにインデックスを利用することが可能になります。これにより、複数の検索条件から取捨選択の上、組み合わせて利用するような要件に対して、作成しなければならないインデックス数の増加を緩和することができます。

```
CREATE INDEX 'ai_airport_day_faa'  
ON 'travel-sample'.inventory.airport(DISTINCT PAIRS({airporname, city, faa}));
```

さらに `self` キーワードを用いることにより、ドキュメントの全てのフィールドを対象とすることができます。

```
CREATE INDEX 'ai_self'  
ON 'travel-sample'.inventory.airport(DISTINCT PAIRS(self));
```

7.6 インデックス物理設計

インデックスの物理設計に関する DDL 構文を紹介し、性能最適化との関係について解説します。

重複 (Duplicate) インデックス

`WITH*1`句を用いて、異なる名前で、共通した定義を持つインデックスを重複して定義することができます。

重複インデックスは、高可用性と負荷分散に役立ちます。

```
CREATE INDEX i1 ON 'travel-sample'.inventory.airport(LOWER(name), id, icao)  
WHERE country = 'France' WITH {"nodes": ["192.0.1.0:8091"]};  
  
CREATE INDEX i2 ON 'travel-sample'.inventory.airport(LOWER(name), id, icao)  
WHERE country = 'France' WITH {"nodes": ["192.0.2.0:8091"]};
```

^{*1} <https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/createindex.html#index-with>

```
CREATE INDEX i3 ON 'travel-sample'.inventory.airport(LOWER(name), id, icao)
WHERE country = 'France' WITH {"nodes": ["192.0.3.0:8091"]};
```

上の例では、3つのインデックスはすべて、同じキーと同じ WHERE 句を持っています。異なる部分としては、インデックスの名前その他、WITH 句で、それぞれ別のノードを指定しています。

WITH 句でノードを指定しない場合であっても、重複した内容を持つインデックスの作成は許されており、その場合、物理配置はランダムに決定されます。

クエリ実行時、重複インデックスはラウンドロビン方式で使用され、負荷が分散されます。

■コラム: エディションによる差異

エンタープライズエディションでは、1つのインデックス定義 DDL を使って、下記のように複数のノード指定を行うことができます。

```
WITH {"nodes": ["node1:8091", "node2:8091", "node3:8091"]}
```

■コラム: WITH 句のその他の利用方法

本書では詳細は触れませんが、WITH 句で指定できる項目には、node の他に、num_replica(インデックスレプリカの数) や(真偽値をとる)defer_build(インデックスの遅延作成) があります。

パーティション

PARTITION BY 句を用いて、インデックスを複数のパーティションに分割することができます。

インデックスのパーティション化には、次のような利点があります。

- **スケールアウト** インデックスサイズの増大に対する、水平方向の拡張による性能最適化が可能です。
- **並行スキャン** 複数のパーティションを同時に並行してスキャン可能なため、クエリ待ち時間の短縮効果が期待できます。
- **透過性** パーティション分割の状態によって、クエリを変更する必要はありません。

ここでは、限られた例を示すに留めますが、例えば、次のインデックス定義 DDL はドキュメントキーでパーティション化されたインデックスを作成します。

```
CREATE INDEX idx ON 'travel-sample'.inventory.airline
(country, name, id)
PARTITION BY HASH(META().id);
```

■コラム: エディションによる差異

インデックスのパーティション化は、エンタープライズエディションでのみ利用できます。

7.7 先行属性マッチと性能最適化

カーディナリティ

一般にデータベースにおけるインデックススキャンについては、カーディナリティが高い（データの種類が多い）項目の検索に向いているということが言えます。これは、カーディナリティが低い（データの種類が少ない）項目の検索では、実際上、フルスキャンに違い動きになる（フルスキャンが向いている）のと裏返しの関係にあります。

Couchbase Server のインデックスにおいても、属性リストの先頭に高いカーディナリティの属性を配置することで、インデックスの検索性能を最適化することができます。ただし、ここで重要なのは、先行属性 (leading attributes) の一致を妨げない限り、という条件が付くことです。

クエリに一致するインデックスの決定は、以下の順序で行われます。

- ① キースペースマッチ
- ② 条件句マッチ
- ③ 先行属性マッチ

具体的には、クエリ中の下記の要素が対応します。

```
SELECT *
FROM retail.east.customer c ①
WHERE c.age = $age ③
AND c.grade = "premium" ②
```

```
CREATE INDEX idx_cust ON retail.east.customer(age, sex)
WHERE grade = "premium"
```

上記のクエリとインデックスの例は、先行属性マッチとカーディナリティの関係が適切な例となります。

ただし、下記のようなクエリでは、上のインデックスを用いることができません。

```
SELECT *
FROM retail.east.customer c
WHERE c.sex = $sex
AND c.grade = "premium"
```

Data サービス API との併用

クエリによる検索の結果として、多数の（かつ大きな）ドキュメントが返されるような場合、クエリではドキュメントキーをクライアントに返し、クライアントから Data サービス API を使用してドキュメントをフェッチすることが考えられます。これにより、Query サービスのプロセスのメモリ領域がボトルネックとなる事を避けることができます。さらに、Couchbase Server では、ドキュメントキーによるデータ取得はそのドキュメントを管理しているノードから直接行われるため、1つの大きなネットワーク伝送を、多数の小さなネットワーク伝送に置き換えることが可能になります。

クエリのサンプルを示します。

```
SELECT META().id as docKey  
FROM retail.east.order  
WHERE META().id IS NOT MISSING  
AND status = "Pending"
```

下記は、上記クエリ実行のためのインデックス定義です。

```
CREATE INDEX idx_order_pending ON retail.east.order(META().id)  
WHERE status = "Pending";
```

上記クエリでは、`META().id IS NOT MISSING` という条件が用いられています。`META().id` を持たないドキュメントは存在しないため、これは一見不要にも思われますが、`META().id` に対する先行属性マッチのために必要になります。

Data サービス API の利用方法については、アプリケーション開発を主題とした最終章の「キーバリュー操作」に関する解説を参照してください。

第8章

アーキテクチャー

8.1 分散アーキテクチャー

分散アーキテクチャーでは、複数のサーバを用いて、1つのプラットフォームを構成します。これをクラスターと呼びます。また、クラスターを構成する各サーバーはノードと呼ばれます。クラスターは、ノード（物理サーバー、仮想マシン、あるいはコンテナ）の追加により、CPU、RAM、ディスク、およびネットワークの容量を増やすことができます。

Couchbase Server は、シェアードナッシング型の分散アーキテクチャーを特徴としており、拡張性と可用性に優れています。

ピアツーピアトポロジー

Couchbase Serverにおいては、他の分散システムに見られるような、マスター／ノード、スレーブノード、構成ノード、ネームノード、管理（ヘッド）ノードなどの概念はなく、全てのノードにおいてソフトウェアは同一です。全てのノードは同等のレイヤーで相互に通信します（ピアツーピアトポロジー）。Couchbase Server は通常、複数のコモディティサーバーから構成されるクラスターとして構成されますが、開発などの目的で、全ての機能を単

一のノードで実行することもできます。その場合であっても、他の分散システムに見られるような、(例えばスタンダードアローンモードのような)複数ノードで構成した場合とは異なる、特別なコンポーネント配置が用いられる訳ではありません。クラスターを单一のノードで構成した場合も、複数のノードで構成した場合も、論理的なアーキテクチャーは変わりません。

Couchbase Server では、ノードがクラスターの一部として構成される(クラスターに参加する)際に、そのノードの特定のサービスを有効または無効にできます(Couchbase Server 管理者は、クラスターへノードを追加する際に、そのノードで実行されるサービスを選択します)。ただし、一旦そのノードがクラスターに参加した後は、そのノードで動いているサービスを変更することはできません。そのようなことを行いたい場合には、一度そのノードをクラスターのメンバーから削除し、改めて参加させる際にサービスを選択し直します。

ピアツーピアレプリケーション

Couchbase Server はアクティブなデータの複製(レプリカ)を自動的に作成し、それらのレプリカをクラスター内のノードに分散して、全てのコピーが別々のノードに配置されるようにします。これは例えば、プライマリサーバーとセカンダリサーバーのように、ノードのレベルで異なる役割を持つアーキテクチャーとは異なっています。Couchbase Server のこのような特徴は、ピアツーピアレプリケーションと呼ばれます。Couchbase Server は、最大 3 つのレプリカをサポートします。

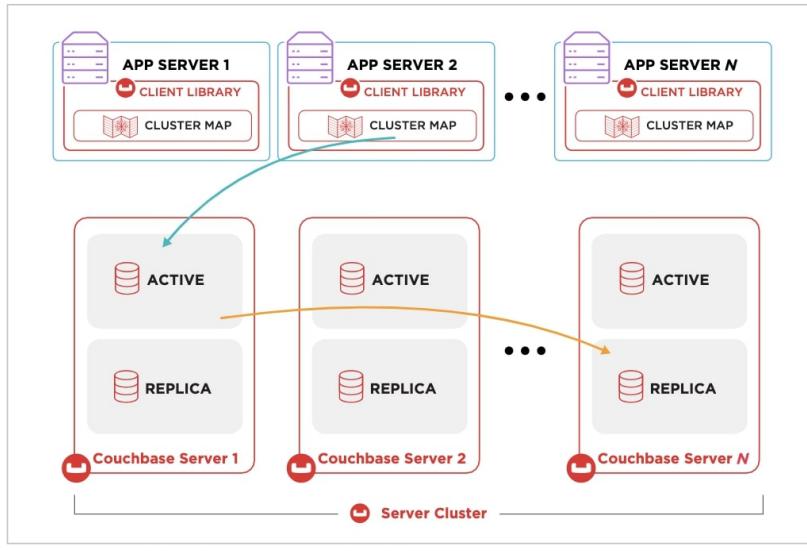
ピアツーピアレプリケーションは、アクティブデータのコピーを他のノードに複製するため、单一障害点を排除します。ノードがダウンした場合、クラスター内の他の場所に存在するレプリカをアクティブ化することによって、そのデータを回復します。このプロセスはフェイルオーバーと呼ばれます。フェイルオーバーは自動または手動で行うことができます。

レプリケーションが提供する冗長性は、ノード障害からデータを保護し、

可用性を向上させます。Couchbase Server では、レプリカは高可用性のみを目的としており、アクティブデータへの昇格が行われない限り、通常は使用されません。これは、分散されたデータを、複数の計算機での分散（並列）処理のために用いる分散システムとは異なっています。そのようなアーキテクチャーで発生する、データの一貫性を維持するための処理に伴うオーバーヘッドは、Couchbase Server では取り除かれています。さらに、後で紹介する、スマートクライアントとの組み合わせにより、高いスループットと低遅延が実現されています。^{*1}

一方、レプリカを取得するための API も提供されており、アクティブデータが利用できなくなつてからフェイルオーバーが実行されるまでの期間、この API を用いることで、データ読み取りの可用性を向上させることができます。

^{*1} 1 ノード当たりの性能として、1 秒間に数十万のリクエストを処理し、かつ、リクエストの 99 パーセントで 1 ミリ秒未満の応答時間が測定されることは珍しくありません。



(画像は、Couchbase Under the Hood: An Architectural Overview^{*2}より引用)

■コラム: レプリカの考え方

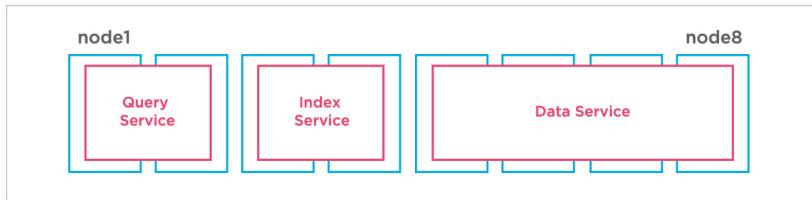
分散アーキテクチャーを持つソフトウェアの中には、例えばレプリカ数が3である（あるいは3レプリケーション）と言った場合、3つのノードにデータが存在することを意味するものがあります。Couchbase Serverでは、レプリカ数が3である場合、アクティブデータを含めて、4つのノードにデータが存在することになります。この「レプリカ」の意味の違いは、前者の種類のソフトウェアでは、複製されたデータの

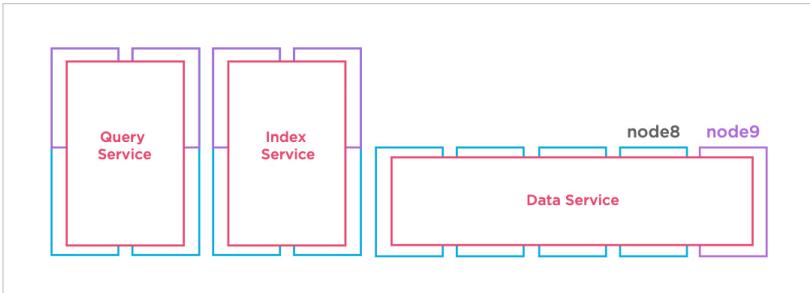
^{*2} https://resources.couchbase.com/c/server-arc-overview?x=V3nd_e

間に区別がないのに対して、Couchbase Server は、アクティブデータとレプリカデータが判然と区別されていることから生じています。

マルチディメンションナルスケーリング (MDS)

ノードは、ドキュメントストレージ (Data サービス)、クエリ (Query サービス)、インデックス (Index サービス) 等、特定のワークロードの処理専用に構成することができます。これによってワークロードを分離し、各サービスの要件と特性に応じたリソース配分にてクラスターを構成することができます。他の分散アーキテクチャーを持つソフトウェアでは、ノードは全て同一のリソースを持つハードウェアで構成されることが必要ないし前提とされているのが屡々見られるのに対して、Couchbase Server は、リソース配分の柔軟性を特徴としており、各サービスに異なるノード数を割り当てるだけでなく、各ノードでワークロード/サービスの特性に応じた異なるリソース (メモリ、CPU) を用いることができます。こうした水平および垂直方向に拡張可能な柔軟なスケーリングは、**マルチディメンショナルスケーリング (MDS)** と呼ばれています。





(画像は、Couchbase Under the Hood: An Architectural Overview^{*3}より引用)

■コラム: エディションによる差異

MDS は、エンタープライズエディションでのみサポートされています。具体的には、コミュニティエディションでは、ノード上のサービスとして Data サービスを無効にすることができます。つまり、エンタープライズエディションでは、Query サービス等を専用のノードを用いて運用することができますが、コミュニティエディションでは、Data サービス以外のサービスは、常に Data サービスと同じハードウェアリソースを共有することになります。

^{*3} https://resources.couchbase.com/c/server-arc-overview?x=V3nd_e

オンラインリバランス

ノードの追加と削除は、データを均等に再分散するリバランスマネージャーを通じて、実現されます。リバランスマネージャーはオンラインで行われ、クライアント/アプリケーションはリバランスマネージャー中も Couchbase Server へ継続してアクセスすることができます。

クラスターは、リバランスマネージャー中もリクエストを処理し続けます。そのため、リバランスマネージャーのためのリソース消費は、クラスターの処理実行に影響を与えます。Couchbase Server では、リバランスマネージャーにおけるデータ移動の並列実行数の設定を通じて、リバランスマネージャーに要する時間とその間のリソース消費とのバランスを最適化することができます。⁴

データベースチェンジプロトコル (DCP)

Couchbase Server の各ノードは、データの複製、インデックスの更新などのために相互に通信します。その際、Couchbase Server は、固有のストリーミングプロトコルであるデータベースチェンジプロトコル (DCP) を用います。DCP は、Couchbase Server で生じたデータの変更をクラスター内部に伝える役割を持ちます。また、DCP は、Spark、Kafka、Elasticsearch のような外部システムとの統合のために利用することができるコネクターへデータをフィードするためにも用いられます。

DCP におけるデータ転送は、メモリーのレイヤーで実行されます。つまり、Data サービスでデータの変更が（メモリレベルで）行われた後、ディスクへの反映を待たずに、DCP を介して他のノードへ転送されます。

DCP は、一時的なエラーに対して堅牢で回復力を持ちます。例えばストリームが中断された場合、接続が再開されると DCP は最後に正常に更新さ

⁴ <https://docs.couchbase.com/server/current/learn/clusters-and-availability/rebalance.html#limiting-concurrent-vbucket-moves>

れた時点から再開します。

vBucket

Couchbase Server は内部的に、**vBucket** を使用して、データを管理しています（クライアント/アプリケーションは vBucket を直接操作しません）。シャードまたはパーティションのようなものといえば、これらの用語に慣れている方はにとっては、理解しやすいかもしれません。あるいは、もっと直接的に、OS 上の物理ファイルに対応しているともいえます。

1 つのバケットは、1024 個のアクティブ vBucket と、(1 レプリケーション毎に) 1024 個のレプリカ vBucket に分割して管理されています。この数はクラスターを構成するノードの数に左右されません。vBucket は、クラスター内で Data サービスを実行しているノード全体に均等に分散されます。Data サービスを実行しているノードの数に増減が生じた場合には、vBucket は再分配されます。

8.2 スマートクライアント

クラスターマップ

Couchbase Server のクライアントは、接続しているクラスターのトポロジー（クラスターの構成）を認識し、トポロジーに変更が発生しても、透過的に最新の状態を認識します。これは、**クラスターマップ**と呼ばれる内部情報が、クラスターから、クライアントへ提供されることによって実現されます。

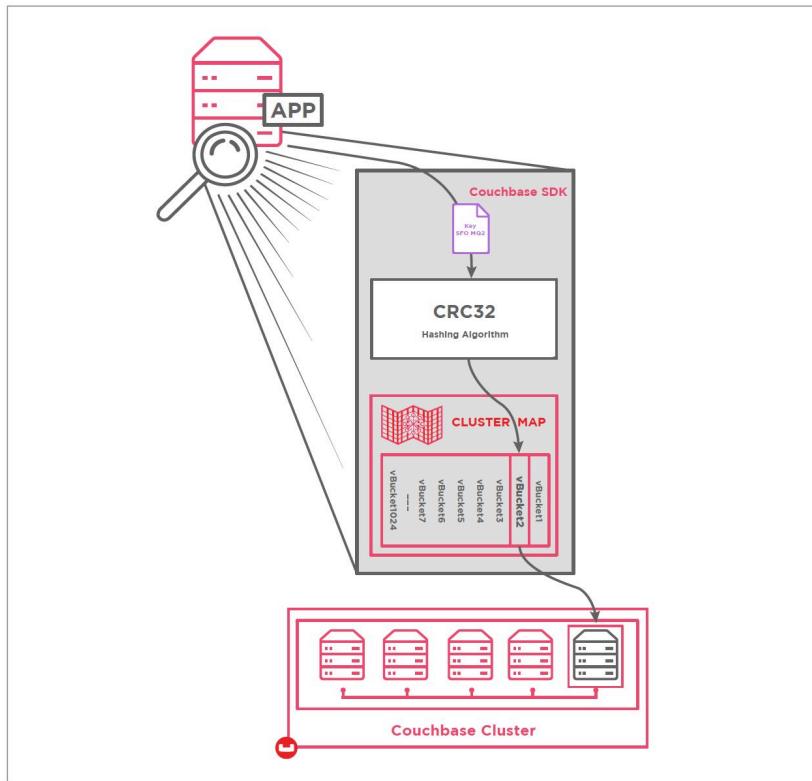
クライアントは、クラスターへの初回接続（ポートストラップ）時にクラスターマップを入手し、クライアントとクラスターとが接続されている間、クラスターマップのローカルコピーは自動的に最新の状態に維持されます。

クライアントは、Data サービスを利用する際、このクラスターマップによって実現されるトポロジー認識に基づいて、目的のドキュメントを管理しているノードに直接アクセスします。

Couchbase Server のドキュメントは、キースペースにおいて、一意のキーにより識別されます。キーは内部的に、CRC32 ハッシュアルゴリズムを用いて変換されます。このアルゴリズムにより、バケットを構成する vBucket 中のドキュメントの分布は、ユーザーの付けたキーの内容に関わらず、均等になります。

Couchbase Server クライアントは、ドキュメントのアクティブなコピーを見つける(ないし、新しく作成する)ために、ドキュメントキーに対して、CRC32 ハッシュアルゴリズムを使用して、そのドキュメントが含まれる(ことになる)vBucket を識別します。次に、クライアントはクラスターマップを参照して、現在どのノードに、その vBucket が含まれているかを判別します。

同様に、クラスターマップでは、Query サービスを実行しているノードのリストも、最新の状態に維持されます。Query サービスはステートレスであるため、任意のノードで任意のリクエストを処理できます。



(画像は、 Moving from Relational to NoSQL: How to Get Started^{*5} より引用)

クラスターマップによるトポロジーの認識は、他の分散アーキテクチャーを持つデータプラットフォームには見られない、Couchbase Server 独自の特徴です。多くの分散アーキテクチャーでは、クライアントとのコミュニケーション

^{*5} <https://resources.couchbase.com/c/relational-no-sql-wp?x=Y7B0ca>

ションを担当する特別なノードが存在し、データが保存されているノードとクライアントとの間を仲介する役割を果たします。この違いは、Couchbase Server が、クライアント/アプリケーションに対して低遅延かつハイスループットな処理を提供することができる要因の1つとなっています。

ブートストラップ

クライアントから Couchbase Server への初回接続は、認証と認可、検出(Discovery)、およびサービス接続の3つのフェーズで確立されます。この一連の過程は、ブートストラップと呼ばれます。

以下に、各プロセスの概要を記します。

1. **認証と認可** クライアントは、ユーザー名とパスワードで認証されます。認証を介して、ユーザーはロールを取得します。ロールに含まれる権限によって、Couchbase Server のリソースに対する操作が認可されます。
2. **検出 (Discovery)** クラスターマップがクライアントに返されます。クラスターマップは、クラスターを構成するノードのリスト、ノード間のサービスおよびデータ配置に関する情報を含んでいます。クライアントは、このクラスターマップを使って、利用するサービスとデータに応じた接続先を検出することができます。
3. **サービス接続** クライアントはサービスレベルの操作を実行するため必要な接続を確立します。クライアントが要求しているリソースへの操作に対する適切な権限に関連付けられているロールを持っている場合、アクセスが許可されます。

なお、ブートストラップ時の接続先は、高可用性を実現するために、クラスターを構成する複数のノードのIPアドレスを用いて構成するのが一般的です。

8.3 XDCR(Xross Data Center Replication)

■コラム: エディションによる差異

コミュニティエディションでは、XDCR は利用できません。

この制限は、バージョン 7.0 より導入されました（バージョン 6.6 までは、一部機能を除き利用できていました）。

エンタープライズエディションでしか利用できない機能は、通常コミュニティエディションとして配布されているバイナリでは、機能自体が含まれていない（実行不可能である）のに対して、XDCR 機能は（少なくとも 7.0 では）表面上、利用可能な状態となっており、規約の上で、利用が制限されています。

XDCR は、なぜ必要か

XDCR は、Couchbase Server クラスター間でデータを複製するために使用されるテクノロジーです。XDCR を使用すると、ベアメタル、VM、プライベートクラウド、パブリッククラウド、コンテナなどのプラットフォームに関係なく、任意のクラスター間でデータを複製できます。また、片方向のデータ同期のみではなく、双方向のデータ同期をサポートしています。

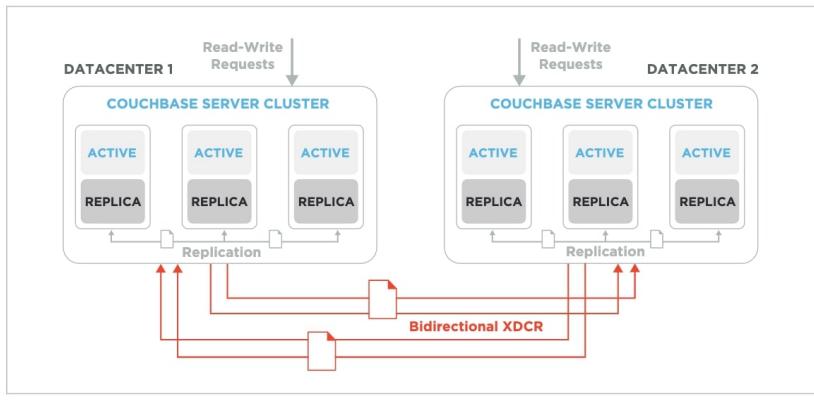
XDCR は、どのように動作するか

XDCR は非同期レプリケーションによる結果整合性を介して、サイト間でデータの一貫性を維持します。

XDCR は、クラスターの構成変更に対して透過的に動作します。ノードの追加ないし削除中であっても、継続して動作し続けます。

第8章 アーキテクチャー 8.3 XDCR(Xross Data Center Replication)

Couchbase Server は、データベースチェンジプロトコル (DCP) と呼ばれるストリーミングプロトコルを持ち、順序付けられたキューを使用してデータの変更を通信します。XDCR はコンシューマーとして、DCP を利用しています。



(画像は、Couchbase Under the Hood: An Architectural Overview^{*6}より引用)

チェックポイント

XDCR は、レプリケーション実行中、一定の間隔でチェックポイントを更新します。レプリケーション中断後の再開時（これには、ユーザーによる計画的な実行、または障害からの回復のケースの両方が考えられます）、XDCR は最後のチェックポイントからレプリケーションを再開します。

チェックポイント更新の間隔が短いほど、レプリケーションの再開に必要な作業量が少なくなります。一方、頻繁なチェックポイントの更新が必要と

^{*6} https://resources.couchbase.com/c/server-arc-overview?x=V3nd_e

第8章 アーキテクチャー 8.3 XDCR(Xross Data Center Replication)

なり、更新のためのシステムリソース消費が、クラスターの性能に影響を与える可能性があります。

レプリケーション指定とフィルタリング

レプリケーションは、バケット単位で設定する他、スコープやコレクションを用いて、制御することができます。

また、レプリケーションの対象とするデータを、データの内容などに応じてフィルタリングするための方法（レプリケーションフィルター）が用意されています。

■コラム: エディションによる差異

コミュニティエディションでは、(XDCR が利用できるバージョンであっても) フィルタリング機能は利用できません。

レプリケーション優先度

あるバケットのレプリケーションが既に実行されている環境で、新しく別のバケットのために新規レプリケーションを追加するようなケースでは、新規レプリケーションの実行が、既存のレプリケーションのスループットに悪影響を与える場合が考えられます。例えば、既存のレプリケーションでは、変更されたデータの更新のみが行われているのに対して、新規にレプリケーションが追加されたバケットに既にデータが格納されている場合、全てのデータのレプリケーションが一斉に実行されることになります。この場合のように、全てのレプリケーションに同じ優先度が適用されることが不適切なケースがあります。

XDCR では、レプリケーション毎に優先度を設定することができ、優先

度に基づいた適切なリソース配分が可能です。

ネットワーク帯域幅節約のための最適化

XDCR は、データ圧縮のような内部機構によって、ネットワークの帯域幅を節約するために最適化されています。また、ユーザーによる設定（例えば、高度な設定^{*7}として提供されている「オプティミスティックレプリケーション」）を介して、環境や用途に応じた最適化を行うことが可能です。

運用上の注意

XDCR 処理は、Data サービスが稼働しているノードで実行されるため、多かれ少なかれ Data サービスに対する性能影響があります。

適切にリソース見積もりが行われることが重要ですが、例えば、運用中のクラスターで、新たに XDCR を利用する要件が生まれた際には注意が必要です。典型的なケースとして、データセンターのマイグレーションを考えられます。状況に応じて、マイグレーションのためにデータのレプリケーションを行う期間、一時的にノードを追加し、クラスターを拡張するといった対応が考えられます。

8.4 サーバーグループ

XDCR が、データセンターレベルで（複数のクラスターを用いて）可用性を高めるために利用できるのに対して、データセンター内の物理配置レベルで（1 つのクラスターを用いながら）可用性を高めるための仕組みとして、サーバーグループが存在します。

データセンター（ネットワーク）内の物理的な区画（ラックやアベイラビリ

^{*7} <https://docs.couchbase.com/server/current/xdcr-reference/xdcr-advanced-settings.html>

ティゾーン等)に合わせて、複数のグループを定義して、サーバーの物理配置に合わせて、ノードをそれぞれのグループに割り当てることができます。

ここでは、2つの異なる区画があり、クラスター内で複製して管理することができる対象(vBucketとインデックス)について、アクティブデータとそのレプリカが1つあると考えてみます。この場合、2つのグループを定義し、物理配置に合わせてノードをそれぞれのグループに割り当てます。サーバーグループの定義によって、アクティブデータと、レプリカとは、別々のグループで維持されます(サーバーグループが定義されていない場合は、配置はクラスター内でランダムに行われます)。そのため、片方の区画の物理的なダメージにより、一方のグループがオフラインになっても、もう一方のグループで、データの完全なセットが維持されます。

■コラム: エディションによる差異

コミュニティエディションでは、サーバーグループは利用できません。

第 9 章

Couchbase Server の構成要素

9.1 Data サービス

Data サービスは、Couchbase Server の管理するデータへのアクセスを提供します。

Data サービスは、全ての Couchbase サービスにおいて、最も基本的なものです。Data サービスは、クラスターを構成する全てのノードのうち、少なくとも 1 つのノードで実行される必要があります。

Data サービスは、以下の 3 つのコンポーネントからなります。

- ディスパッチャー (Dispatcher)
- KV(キーバリュー) エンジン (KV Engine)
- スケジューラー (Scheduler)

ディスパッチャー

- Data サービスへのリクエストを受け取り、レスポンスします。

- DCP(Database Change Protocol) を使用して、クラスター内の他のノードおよび他のクラスターにデータをストリーミングします。
- 認証を処理します。

KV エンジン

バケットに関する以下の機能を提供します。

- マネージドキャッシュは、設定されたクォータに従って、バケットに割り当てられたメモリです。アイテムが書き込まれると、アイテムはまずキャッシュに入り、その後、1つ以上の他のノードに複製するために、複製キューに配置されます。さらに（揮発性の Ephemeral バケットではなく、Couchbase バケットの場合）ディスクキューを介して、ディスクに書き込まれます。また、クラスター全体のさまざまなノード上のバケットアイテムの場所が記録されている、パーティションハッシュテーブルも、このマネージドキャッシュに含まれます。
- チェックポイントマネージャーは、アイテムの変更を追跡します。チェックポイントには、まだレプリケーションキューとディスクキューに配置されていないメモリ内の変更が記録されます。
- アイテムページヤーは、最近使用されていないアイテムを、必要に応じてメモリから取り除き、スペースを解放します。
- フラッシャーは、バケット内のすべてのアイテムを一括削除することができます。バケットのデフォルト設定では、フラッシュ処理は無効になっています。
- 有効期限 (Expiry) ページヤーは、期限切れのアイテムをスキャンし、メモリとディスクから消去します。有効期限ページヤーは、デフォルトで 60 分ごとに実行されます。
- バッチリーダーは、ディスクキューに溜まっている複数のアイテムに加えられた変更を、ディスクに書き込むために、バッチとして結合す

ることにより、パフォーマンスを向上させます。

スケジューラー

Data サービスのスケジューラーは、以下の 4 種類のスレッドからなります。これらのスレッドは、互いに独立して、影響を与えずに実行されます。

- **Reader IO** ディスクから情報を読み取るスレッド
- **Writer IO** ディスクに情報を書き込むスレッド
- **Non IO** ディスクアクセスを必要としない処理 (接続通知、チェックポイントの削除、ハッシュテーブルのサイズ変更など) を行うスレッド
- **Aux IO (補助 auxiliary IO)** 補助的な処理 (フェッチ、スキャン、バックフィルタスク等) を行うスレッド

9.2 Query サービス

Query サービスは、N1QL クエリ言語によるクエリをサポートします。

Query サービスは次の 3 つの構成要素からなります。

- リスナー
- クエリプロセッサ
- データストア

リスナー

クエリ要求を受信します。

クエリプロセッサ

クエリプロセッサは、受信されたクエリが有効なステートメントであるかどうかを判定するために、クエリにパーサーを適用します。そして、利用可能な実行パスを評価するオプティマイザーを用いて、待ち時間が最も短いパスを決定します。さらに、決定されたパスを使用してクエリ実行プランを生成し、計画を一連の操作（オペレータ）にまとめます。最後に、実行エンジンがオペレータを受け取り、実行します。

データストア

データソースへのアクセスを提供します。データソースには、Couchbase Server のデータ、インデックス、認証情報等が含まれます。

他のサービスとの連携

Query サービス内部の処理は、他のサービスと連携しながら、以下のように行われます（可能な場合、処理は並行で実行されます）。

1. クエリプロセッサによる解析ルーチンを実行して、リクエストされたステートメントを検証の上、実行プランを作成します。
2. Index サービスにアクセスして、関連するインデックスに対してスキャン操作を実行します。
3. Data サービスにアクセスして、フェッチ操作を実行し、返されたデータを結合操作で使用します。
4. フィルタリング、集約、ソートなどの追加の処理を実行します。

■コラム: エディションによる差異

コミュニティエディションではクエリの並列実行におけるリソース利用は、最大4コアに制限されているのに対して、エンタープライズエディションではこの制限がありません。

このことは、コミュニティエディションでは、ノード上の Data サービスを無効にすることができないため、Query サービスと Data サービスは、常に同じハードウェアリソースを共有している、という事実を考え合わせて理解することができます。

9.3 Index サービス

Index サービスは、Query サービスと連携して N1QL クエリを実行するためのインデックス（データへの索引）機能を提供します。

Index サービスは、以下の3つのプロセスからなります。

- スーパーバイザー
- プロジェクター
- ルーター

スーパーバイザー

スーパーバイザーは Index サービスのメインプログラムです。Index サービスノードで稼働し、以下の役割を持ちます。

- インデックス定義の管理
- インデックス作成・保存
- インデックス更新
- Query サービスからのリクエストへの応答

インデックス作成時には、まず Query サービスから、スーパーバイザーに対して、インデックス作成リクエストが送られます。その後、スーパーバイザーは、インデックス定義をプロジェクトとルーターに渡します。

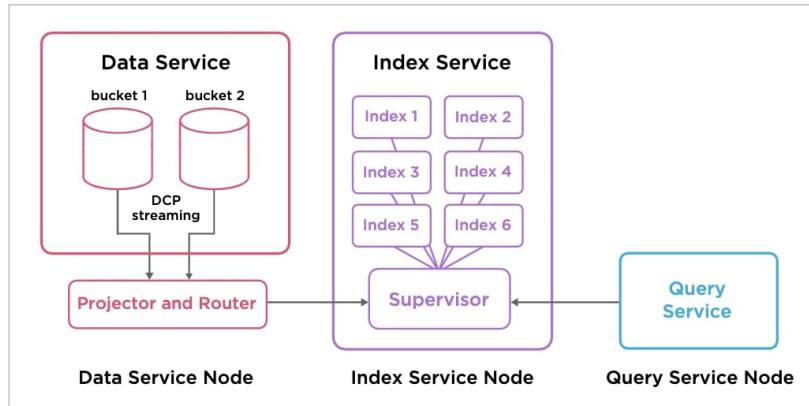
プロジェクトとルーター

プロジェクトとルーターは Data ノード上で実行されるプロセスです。スーパーバイザーによって提供されるインデックス定義に従って、Index サービスにデータを提供します。

プロジェクトとルーターは、Data サービスによって提供される DCP ストリームを継続的に監視します。DCP ストリーム上のデータミューション（更新）情報が、インデックスのターゲットを含む場合には、プロジェクトとルーターが、そのデータをスーパーバイザーに送信して、スーパーバイザーがインデックスを更新します。

インデックスが作成されていない場合、プロジェクトとルーターは何もしません。

以下は、Index ノードのスーパーバイザー、Data ノードのプロジェクトとルーター、および Query サービスとの間のコミュニケーションの概念図です。



(画像は、Couchbase Under the Hood: An Architectural Overview^{*1}より引用)

9.4 クラスターマネージャー

クラスターマネージャーは、クラスターを構成する全てのノード上に存在します。

クラスターマネージャーは次の2つのプロセスで構成されています。

- ns-server
- babysitter

^{*1} https://resources.couchbase.com/c/server-arc-overview?x=V3nd_e

ns-server

ns-server は、クラスターマネージャーの中心となるプロセスであり、以下の機能を持ちます。

- **マスターサービス** フェイルオーバー、リバランス、バケットの追加と削除など、クラスター全体に影響を与える操作を管理します。常に、クラスター上のノードの 1 つだけがマスターサービスの役割を担当します。マスターサービスは、ノード間でのネゴシエーションの上、決定されます。選出されたノードが使用できなくなった場合は、別のノードが引き継ぎます。マスターサービスは、オーケストレーターと呼ばれることがあります。
- **REST API 管理** クラスター管理に関する REST API サービスを提供します。この REST API は、Web コンソールで提供される機能とコマンドラインインターフェイスの両方の基礎になっています。Couchbase Server では、REST API インターフェースを受け持つ特別なノードはありません。
- **認可** ロールベースのアクセス制御でノードのリソースを保護します。

babysitter

babysitter は、Couchbase Server の様々なプロセスの保守を担当します。保守の対象には、クラスターマネージャーのもう 1 つのプロセスである ns-server を含みます。

babysitter は、以下の役割を持ちます。

- プロセスを監視し、監視ログ出力を babysitter.log に記録します。
- プロセスのいずれかが停止した場合、そのプロセスを起動します。

9.5 マスターサービス

マスターサービスとして選出されたクラスターマネージャーは、独自の役割を果たします。

マスターサービスの役割は、下記のように大別されます。

- **サービス管理** ノードの現在の状態を管理し、プロセスとサービスの監視と再起動を処理します。
- **バケット管理** バケットレベルの操作、レプリケーション、フェイルオーバー、再起動、および統計収集などを処理します。

以下に、マスターサービスが行ういくつかの処理の概要を記します。

クラスタートポロジー管理

マスターサービスは、クラスターのメンバー構成に責任を持ちます。クラスターのトポロジーが変更されると、既存のワークロードの処理を継続しながら、クラスターの再構成を行います。

クラスターに新しく Data ノード (Data サービスが稼働しているノード) が追加される際には、マスターサービスがリバランスを開始し、新しい vBucket マップレイアウトを計算します。その後、新しい vBucket マップレイアウトに従って、既存のノードから追加されたノードへの vBucket のコピーが開始されます。vBucket のコピーが完了すると、古い vBucket から新しい vBucket へのスイッチオーバーが発生し、新しい vBucket がアクティブになると、マスターサービスは、新しい vBucket マップをすべてのノードとクライアントに通知します。

Data ノードをクラスターのメンバーから取り外すプロセスでは、維持されるノードで新たに vBucket が作成され、削除されるノードからデータが移動されます。削除対象ノードからデータがなくなった後に、そのノードは

クラスターから削除されます。

ノード障害検出

Couchbase Server クラスター内のノードは、ハートビートメカニズムによってステータスが管理されます。ハートビートは、全てのノード上のクラスマネージャーから定期的に提供されます。各ハートビートには、ノードの状態を評価するために使用される基本的な情報が含まれています。

マスターサービスは、他のすべてのノードから受信したハートビートを追跡します。自動フェイルオーバーが有効になっていて、デフォルトのタイムアウト期間より長くノードからハートビートが受信されない場合、マスターサービスはノードをフェイルオーバーの対象としてマークし、実行可能な条件が整っている場合、自動フェイルオーバーを行います。

バケット管理

マスターサービスは vBucket ファイルの配置をコントロールします。vBucket の配置は、クラスタートポロジーが変更されるたびに、次のルールによって再編成されます。

- アクティブ vBucket とレプリカ vBucket は別々のノードに配置されます。
- 2 個以上のレプリカ構成の場合、レプリカ vBucket はそれぞれ別々のノードに配置されます。
- サーバーグループが定義されている場合、アクティブ vBucket とレプリカ vBucket は別のグループに配置されます。

管理、統計、およびロギング

クラスターマネージャーは、構成管理、統計収集、およびロギングサービスを一元化し、管理を簡素化します。すべての構成変更はマスターサービスによって管理され、マスターサービスノードから他のノードに通知されます。

第 10 章

セキュリティ

10.1 主要なセキュリティ要素

はじめに、Couchbase Server におけるセキュリティ要素の概観を示します。^{*1}

認証 (Authentication)

Couchbase Server へのユーザー (管理者や、アプリケーション) からのアクセスは、認証を介してのみ行うことができます。ユーザーは、ローカルまたは外部のパスワードレジストリを使用して認証されます。認証には、資格情報をサーバーに直接渡す方法と、資格情報が埋め込まれているクライアント証明書を使用する方法があります。

認可 (Authorization)

Couchbase Server は、ロールベースのアクセス制御 (RBAC) によって、ユーザーを、個別に割り当てられた役割に関連付けます。ロールは、特定の

^{*1} <https://docs.couchbase.com/server/current/learn/security/security-overview.html>

システムリソースへの、特定の操作を許可する権限のセットです。これにより、ユーザーがアクセスしようとしているシステムリソース、および実行しようとしている操作に応じて、システムはそのアクセスを許可または拒否します。

ロールは、ユーザー認証を介してクライアントに付与されます。

ロールは、ユーザーおよびグループに対して割り当てることができます。

■コラム: エディションによる差異

コミュニティエディションには、ユーザーをグループ化する機能はなく、ロールはユーザーのみに割り当てることができます。

コミュニティエディションにおけるロールの位置づけは、限られたビルトインロールの提供に留まっており、本格的なロールベースのアクセス制御 (RBAC) とは異なります。

監査 (Auditing)

Couchbase Server で実行されるアクション (発生するイベント) を監査 (ログ出力) することができます。

監査イベントの種類については、リファレンス^{*2}を参照することができます。

^{*2} <https://docs.couchbase.com/server/current/audit-event-reference/audit-event-reference.html>

暗号化 (Encryption)

Couchbase Server が管理するデータは、適切な復号化手段を所有する許可された関係者による場合を除き、読み取り不可能になるように暗号化することができます。これによって、データを安全に保存または送信することができます。

10.2 セキュリティ管理項目

Couchbase Server のセキュリティを高めるために、管理可能な領域について紹介します。^{*3}

ネットワークアクセス

暗号スイート、TLS レベル、およびコンソールアクセスを、個別に管理することができます。

クラスターに対するアクセスのみではなく、クラスターを構成するノード間の通信の保護も可能です。

証明書

クラスターに対して、証明書を登録できます。

クライアントによって提示された証明書を許可することができます。

ログリダクション

個人情報がログに含まれないようにすることができます。

^{*3} <https://docs.couchbase.com/server/current/manage/manage-security/security-management-overview.html>

セッションタイムアウト

一定期間非アクティブな状態が続いた後に、セッションを終了するように構成することが可能です。

ポートマッピング

適切なファイアウォール設定を行うために、Couchbase Server が利用する必要のある全てのポート番号の情報が公開されています。^{*4}

これらのポートの多くについて、マッピングを変更することが可能です。

■コラム: エディションによる差異

以下のセキュリティ関連機能は、エンタープライズエディションの機能として提供されています。

- 外部アクセシビリティ管理
- ロールベースアクセスコントロール (RBAC)
- ネットワークアクセス暗号化
- X.509 証明書
- 監査
- ログリダクション
- ノード間暗号化
- サイファー管理

^{*4} <https://docs.couchbase.com/server/current/install/install-ports.html>

10.3 認証

認証 (Authentication^{*5}) は、誰がシステムにアクセスしようとしているのかを識別するためのプロセスです。Couchbase Server にアクセスするには、認証される必要があります。

Couchbase Server の認証は通常、ユーザー名とパスワードに依存します。これらの情報は、アクセスを試みるユーザー（管理者やアプリケーション）から渡される必要があります。ユーザー名とパスワードは、事前に定義されているものと一致する必要があります。事前の定義情報は、Couchbase Server クラスター、または外部の認証システムのいずれかで管理されます。

また、クライアントアプリケーションから、X.509 証明書を使用して Couchbase Server に資格情報を渡すこともできます。

ビルトイン管理者アカウント

クラスターの初期化時に、ビルトイン管理者アカウントのユーザ名とパスワードを設定します（その後、任意の数のユーザーをクラスターにいつでも追加できます）。

ビルトイン管理者アカウントとは別に、管理者権限を持つユーザーを作成することは可能ですが、ビルトイン管理者アカウントとクラスター初期化後に追加作成されたユーザーとでは、管理方法が異なります。ビルトイン管理者アカウントのパスワードを変更する際には、サーバー上で下記のコマンドを実行する必要があります。

```
couchbase-cli reset-admin-password [--regenerate] [--new-password <password>] [--port <port>]
```

^{*5} <https://docs.couchbase.com/server/current/learn/security/authentication.html>

パスワードポリシー設定

Couchbase Server のパスワードポリシーを、CLI を使って変更することができます。

下記は、その実行例です。

```
couchbase-cli setting-password-policy  
--cluster http://localhost:8091 -u Administrator  
- p password --set --min-length 8 --uppercase  
--lowercase --digits --special-chars
```

デフォルトのポリシーは、以下の通りです。

```
{  
  "enforceDigits": false,  
  "enforceLowercase": false,  
  "enforceSpecialChars": false,  
  "enforceUppercase": false,  
  "minLength": 6  
}
```

外部アクセシビリティ管理

外部システムで認証されるユーザーを、クラスターに追加することができまます。外部ユーザーは外部システムで認証されるため、クラスターでは認証情報(パスワード)を管理しません。

外部システムによるアクセシビリティ管理としては、次のいずれかが用いられます。

- LDAP (Lightweight Directory Access Protocol)
- PAM (Pluggable Authentication Modules)

■コラム: エディションによる差異

外部アクセシビリティ管理は、エンタープライズエディションでのみ利用可能です。

Web コンソールでの認証

Web コンソールへのログインがデフォルトのポート (8091) による http アクセスで実行されている場合、ユーザー名とパスワードは、平文で渡されます。

必要に応じて、セキュアなアクセス (ポート 18091 による https アクセス) を利用するように構成し、ユーザー名とパスワードを暗号化された形式で渡すようにすることができます。

アプリケーションの認証

アプリケーションから資格情報を渡すには、SASL(Simple Authentication and Security Layer) フレームワークによって提供される 4 つのメカニズムのいずれかを使用します。

このパスワード認証メカニズムには、次の方があります (強度による昇順で掲載)。

- **PLAIN** 暗号化されていない形式でパスワードを送信します。

- **SCRAM-SHA1** 160 ビットのキーを使用します。
- **SCRAM-SHA256** SHA2 と呼ばれるハッシュ関数グループの 1 つ。256 ビットのキーを使用します。
- **SCRAM-SHA512** SHA2 グループのハッシュ関数。512 ビットのキーを使用します。サポートされている最も強力な認証プロトコルです。

SCRAM(Salted Challenge Response Authentication Mechanism) を用いて、保護された形式でパスワードを送信することにより、アプリケーションを安全に認証できます。

クライアント/サーバー間ネゴシエーションにより、クライアント(アプリケーション)とサーバー(Couchbase Server)の両方の OS でサポートされている最も強力な認証プロトコルが選択されます。たとえば、クライアントが PLAIN プロトコルのみをサポートしている場合、PLAIN プロトコルが使用されます。クライアントが SCRAM-SHA1 プロトコルをサポートしている場合は、SCRAM-SHA1 が使用されます。

SCRAM チャレンジ/レスポンスプロトコルは、パスワード検証のプロセスのみを保護します。後続のセッションを保護するには、TLS を使用する必要があります。

証明書ベースの認証

Couchbase Server は、クライアント認証のための X.509 証明書の使用をサポートしています。⁶

⁶ <https://docs.couchbase.com/server/current/learn/security/certificates.html>

10.4 認可

認証が成功した後、認可 (Authorization^{*7}) が行われます。認可 (または承認と呼ばれる場合もあります) により、適切なアクセスレベルが決定されます。

データベースユーザーの種類

ロール (Role^{*8}) について説明するにあたって、まず基本的なユースケースを整理します。一般にアプリケーションのバックエンドとして使われるデータベースのユーザーは、管理者、開発者、アプリケーションに分類することができます。

- 管理者は、クラスターの管理タスクを担います。Web コンソールにログインしたり、CLI を使って管理タスクを実行します。データの登録を担当することも考えられます。
- 開発者は、データを参照したり、データの書き込みを行うためにデータベースにアクセスすることがあります。また、Web コンソールなどでクラスターの情報を参照する場合があります。
- アプリケーションは、データベースに対して、データの読み取りと書き込みを行います。

ビルトインロール

以下に、ビルトインロールの概要を記します。

^{*7} <https://docs.couchbase.com/server/current/learn/security/authorization-overview.html>

^{*8} <https://docs.couchbase.com/server/current/learn/security/roles.html>

- **Full Admin** ロールは、Couchbase Server の全ての機能やリソースへの完全なアクセスを許可します。
- **Read-only Admin** ロールは、統計情報を含むクラスター管理情報の参照を許可します。また、Web コンソールへのアクセスを許可します。
- **Application Access** を介して、特定のバケットに対して、全ての権限を許可します。

■コラム: エディションによる差異

エンタープライズエディションでは、よりきめ細かいロールベースのアクセスコントロール (RBAC) を利用可能です。

10.5 暗号化

Couchbase Server は、データの暗号化 (Encryption^{*9}) と復号化を、様々なレベルで広範囲にサポートします。

通信データ

通信データ (On the Wire) 暗号化のために、TLS が利用できます。

通信データ暗号化の範囲は、クライアント/サーバー間、コンソールアクセス、ノード間、およびクラスター間 (XDCR) をカバーします。

また、TLS のための X.509 証明書の利用をサポートしています。

^{*9} <https://docs.couchbase.com/server/current/learn/security/encryption-overview.html>

永続化データ

永続化データ (At Rest) 暗号化の機能は、下記のようなサードパーティー暗号化ツールとの組み合わせにより実現されます。

- Linux Unified Key Setup (LUKS)^{*10}
- Thales CipherTrust^{*11} (旧 Vormetric/Gemalto)
- Protegrity^{*12}

LUKS による暗号化については、設定手順についてのドキュメント^{*13}とスクリプト^{*14}が提供されています。

システム機密情報

データベースに登録されているパスワードや証明書のような、システム機密情報 (System Secrets^{*15}) は、暗号化されたフォーマットでディスクに保存されます。

他のデータベースには、ユーザーの情報をデータベース (システムテーブル) で管理し、参照することのできるものもありますが、Couchbase Server は、それらとは異なり、ユーザー情報の管理・参照のために、システムテーブルは利用されていません。

^{*10} https://en.wikipedia.org/wiki/Linux_Unified_Key_Setup

^{*11} <https://cpl.thalesgroup.com/encryption/transparent-encryption>

^{*12} <https://www.protegrity.com/>

^{*13} <https://docs.couchbase.com/server/current/manage/manage-security/manage-connections-and-disks.html#luks-encryption-procedure>

^{*14} https://github.com/couchbase/perfrunner/blob/master/scripts/create_luks_fs.sh

^{*15} <https://docs.couchbase.com/server/current/manage/manage-security/manage-system-secrets.html>

JSON ドキュメントフィールド

Couchbase Server Java SDK を使用して、JSON ドキュメントの特定のフィールドを暗号化および復号化することができます。

フィールドレベルの暗号化 (Field Level Encryption^{*16}) の操作方法として、次の 2 つの方法が利用可能です。

- データバインディング中の透過的な暗号化/復号化
- フィールドに対する直接的な操作

■コラム: エディションによる差異

フィールドレベルの暗号化は、エンタープライズサブスクリプションライセンス契約により提供されるオプションのライブラリとして実現されています。

^{*16} <https://docs.couchbase.com/java-sdk/current/concept-docs/encryption.html>

第 11 章

フェイルオーバー

11.1 フェイルオーバー概観

Couchbase Server におけるフェイルオーバーの解説に進む前に、まず分散アーキテクチャーにおけるフェイルオーバーについて、概観します。

まず、広い意味でのフェイルオーバーとして、以下の 2 種類があります。

計画的なノード削除（グレースフルフェイルオーバー）

クラスターから、機能しているノードを削除するには、まず（管理コンソール等を利用し）ノードを削除対象としてマークします。その後、クラスターをリバランスし、これまで、そのノードへ行われていたリクエストが他のノードで処理できるようにします。

削除対象ノードは、リバランスが完了するまでデータリクエストを処理することができます。リバランス完了後に、他のノードでリクエストが処理されることになります。すなわち、サービスが中断することも、データを消失してしまうこともなく、グレースフルに構成を縮小（Graceful degradation^{*1}）することができます。

^{*1} https://en.wikipedia.org/w/index.php?title=Graceful_degradation&redirect=no

障害時のノード削除（ハードフェイルオーバー）

ハードフェイルオーバーは、計画的なノードの削除とは異なり、正常に動作していないノードに対して実行します。ノード障害時に他のノードへ複製されていないデータが存在していた場合、データロスとなります。

機能しているノードをハードフェイルオーバーした場合、ハードフェイルオーバーは即座にノードをクラスターから削除するため、データロスが発生する可能性があります。

以下、上記のハードフェイルオーバーに関する論点を整理していきます。

フェイルオーバー実行方法と考慮点

ノード障害発生時には、クラスターを縮退稼働させるために、フェイルオーバーを手動で実行するか、あるいは、ソフトウェアにより自動的にフェイルオーバーをトリガーすることが考えられます。フェイルオーバーはクラスターの性能を縮小させてしまうため、状況をどのように処理すべきかはよく検討されるべきです。

- **自動フェイルオーバー**を利用すると、ユーザーによる操作なしにノードをフェイルオーバーできますが、ノード障害を発生させた問題の調査や特定は難しくなります。
- **手動フェイルオーバー**によりクラスターを管理する場合、クラスターを監視し、問題の発生を検知できるようにします。問題が発生したら、手動でフェイルオーバーを実行します。この手法では、より多くの監視や手動の操作が必要になります。

自動フェイルオーバーにおける問題

問題の原因を特定できない場合、または残りのシステムにかかる負荷を理解していない場合、自動フェイルオーバーは、より多くの問題を発生させる可能性があります。例えば、次のような状況を考えてみることができます。

クラスターが、キャパシティの限界付近（例えば、80%から90%）で稼働しているとします。この時ノード障害が発生し、ソフトウェアが自動的にノードをフェイルオーバーしたとします。残りのノードでは負荷が増大し、正常に処理を続けることはできないでしょう。結果として、負荷の増大により、他のノードもダウンし、自動的にフェイルオーバーされてしまいかねません。このように、障害が連鎖し、結果的にクラスター全体がダウンしてしまう可能性があります。

この場合に考えられるのは、單一ノード障害が発生してもクラスターを運用し続け、準備が整い次第、新しいサーバーをクラスターに追加し、キャパシティを補填した上で、ダウンしたノードを削除し、リバランスを実行することです。こうすることによって、クラスター全体が利用不可能になる可能性を回避することができます。クラスター全体の障害によって、まったくリクエストを処理できなくなるよりも、一部のリクエストを処理できない方が、良い場合があります。

もちろん、ノード障害発生時にも十分な余剰のキャパシティを確保し、縮退運転ができるようにしておくことが望ましいのは間違ひありません。

自動フェイルオーバーにおける制約

分散アーキテクチャーを持つソフトウェアの多くでは、以下の制約付きで自動フェイルオーバーを実装しています。

- 自動フェイルオーバーは、最低でも1クラスターに3台のノードを必要とする。これにより、ネットワーク分断が発生した際に、お互い

のノードをフェイルオーバーしようとする状況が回避されます。これは、データの整合性と一貫性を保護するために重要です。

- ・自動フェイルオーバーは、データの複製が存在するなど、発動後もサービスが継続可能な場合のみ実行される。
- ・自動フェイルオーバーは、管理操作を必要とする前に一度だけ（または、余剰のキャパシティが十分にあることが想定された元で、規定回数まで）発動する。これは、フェイルオーバーの連鎖による、性能や安定性の劣化を防ぎます。
- ・自動フェイルオーバーを実行するまでに一定の待機時間を設ける。これは一時的なネットワーク障害や、システムの遅延によって、誤ってノードがフェイルオーバーされることを防ぎます。

保守運用

保守要員による運用では、アラートに対する次の行動に関する意思決定者の役割が重要になります。適切な人員配置により、広範囲のデータ、観測、経験を生かして、最適な方法で状況を解決することができます。

組織において、しかるべき担当者による影響関係に対する判断を伴わない、システムによるフェイルオーバーの自動化が許可されていないことも珍しくありません。

外部システム連携

クラスターをモニタリングし、一定の発動条件に基づき、フェイルオーバーを実行するために、外部システムを利用することも考えられます

外部システム利用には、ソフトウェアによる自動発動に比べて、クラスター以外の環境に関しても考慮できるという面での優位性があります。例えば、外部システムは、クラスターが依存するネットワークスイッチがダウンしていることを検知するかもしれません。このような場合、ノードをフェイ

ルオーバーしても状況は改善しないことが分かるため、フェイルオーバーを実行しないという判断が可能になります。

モニタリングシステムにより、問題は単一のノードだけで起きており、クラスターを縮退させても、残りのノードで集約したトラフィックを処理できると判断できた場合にのみ、フェイルオーバーを実行することも考えられます。

外部システムからフェイルオーバーを実行する場合、REST API やコマンドラインツールを利用することが考えられます。

11.2 自動フェイルオーバー

Couchbase Server における自動フェイルオーバー (Automatic Failover^{*2}) について、概略を整理します。

有効化設定

Couchbase Server の自動フェイルオーバーは、デフォルトでは無効となっています。これは、自動フェイルオーバーの発生による影響関係を理解した上で、明示的に有効とされるまで、自動フェイルオーバーが発生することを防止するためです。

実行条件

自動フェイルオーバー実行には、いくつかの条件があります。これは自動フェイルオーバーが実行された際に発生する可能性のある問題を回避するためです。

下記のようなケースでは、自動フェイルオーバーが発生しません。

^{*2} <https://docs.couchbase.com/server/current/learn/clusters-and-availability/automatic-failover.html>

- **イベント同時発生** 複数のイベントが同時に発生した場合、自動フェイルオーバーはトリガーされません。
- **フェイルオーバー連続実行** 管理者が指定した規定回数までしか、自動フェイルオーバーはトリガーされません。規定回数として許可される最大値は3です。自動フェイルオーバーの実行が規定回数に達すると、管理者がカウントを手動でリセットするまで、自動フェイルオーバーはトリガーされません（規定回数に達する前にカウントを手動でリセットできます）。
- **データ損失発生可能性** データ損失が発生する可能性のある状況では自動フェイルオーバーはトリガーされません。たとえば、残りのノードにデータの複製がない場合はこのケースです。
- **非フェイルオーバーノードの応答状況** フェイルオーバー対象となるノードを除いた後でなお、クラスターを構成するノードの過半数からの応答がある場合のみ、自動フェイルオーバーがトリガーされます。

待機時間

ノードが自動フェイルオーバーされるまでに、最低でも30秒の待機時間が必要とされます。適切な待機時間を設定することは、サーバーやネットワークの一時的な問題により、機能しているノードをフェイルオーバーしてしまうことを防ぐために重要です。

通知

ノード障害が発生し、自動でフェイルオーバーされた際にメールで通知を送信するように設定することができます。

サービス固有ポリシー

自動フェイルオーバーの発現条件の1つとして、フェールオーバー対象ノード上で稼働しているサービス固有のポリシーがあります。

Couchbase Server では、マルチディメンションナルスケーリング (MDS) により、ノード毎に配置するサービスを選択することが可能です。フェイルオーバーの挙動は、そのノード上で稼働しているサービスが自動フェイルオーバーに対応しているかどうかにより決定されます。

- **Data** サービスでは、自動フェイルオーバーが有効化されるためには、最低3ノードが必要です。
- **Index** サービスでは、自動フェイルオーバーはサポートされません。
- その他のサービス (Query、Service、Analytics、Eventing) のフェイルオーバーには、最低2ノードが必要です。

■コラム: エディションによる差異

コミュニティエディションでは、マルチディメンションナルスケーリング (MDS) は利用できないため、Data サービスのポリシーに準拠することになります。

グループフェイルオーバー

サーバーグループを定義し、ノードをグループ化することにより、グループフェイルオーバーの機能を活用することができます。

グループフェイルオーバーは、デフォルトでは無効とされています。

グループに多数のノードが含まれている場合でも、グループフェイルオーバーは単一のイベントと見なされます。

■コラム: エディションによる差異

グループフェイルオーバーは、エンタープライズエディションでのみ使用できます。

ディスク障害に対するフェイルオーバー

ディスク障害に対する自動フェイルオーバーの機能を、クラスター単位で有効化することにより、活用することができます。

■コラム: エディションによる差異

ディスク障害に対するフェイルオーバーは、エンタープライズエディションでのみ使用できます。

11.3 手動フェイルオーバー

Couchbase Server のフェイルオーバーは以下の方法で手動で実行できます。

Web コンソール

Web コンソールの Servers メニューに移動します。ノード一覧が表示され、クラスターからダウンしていると判定されているノードのみ「Fail Over」ボタンが有効になります。フェイルオーバーしたいノードの「Fail Over」ボタンをクリックし、ノードをフェイルオーバーします。

コマンドラインツール

`couchbase-cli` の `failover` サブコマンドを利用して、ノードをフェイルオーバーすることができます。³

ノードをフェイルオーバーするには、フェイルオーバーするノードの IP アドレス（および標準ポート番号を利用していない場合はポート番号）を指定します。

```
couchbase-cli failover --cluster=<server(available)>:8091\  
-u cluster-username -p cluster-password\  
--server-failover=<server(target)>:8091
```

REST API

REST API を利用して、フェイルオーバーを実行することができます。⁴ 以下に、`curl` コマンドを用いた呼び出し方法を示します。

³ <https://docs.couchbase.com/server/current/cli/cbcli/couchbase-cli-failover.html>

⁴ <https://docs.couchbase.com/server/current/rest-api/rest-node-failover.html>

```
curl -v -X POST -u <username>:<password>
      http://<ip-address-or-hostname>:<port>/controller/failOver
      -d [otpNode=node_1@<hostname>]
      -d allowUnsafe=< true | false >
```

■コラム: Couchbase Server におけるグレースフルフェイルオーバーとハードフェイルオーバー

Couchbase Server では、手動でフェイルオーバーを実行する場合(つまり、Web コンソール、コマンドライン、REST API のいずれにも)、グレースフルフェイルオーバー (Graceful Failover^{*5}) とハードフェイルオーバー (Hard Failover^{*6}) の両方の手段が用意されています。

グレースフルフェイルオーバーは、応答しているノードをクラスターから削除する際に用いられます。その場合、Couchbase Server には、クラスターの削除という手段も存在します。それぞれの長所と短所を含め、グレースフルフェイルオーバーについて、さらに理解するために、ドキュメント^{*7}を参照することができます。

ハードフェイルオーバーについては、応答していないノードをクラスターから削除する際に用いられます。さらなる理解のために、ドキュメント^{*8}を参照することができます。

-
- *⁵ <https://docs.couchbase.com/server/current/manage/manage-nodes/failover-graceful.html>
 - *⁶ <https://docs.couchbase.com/server/current/manage/manage-nodes/failover-hard.html>
 - *⁷ <https://docs.couchbase.com/server/current/learn/clusters-and-availability/graceful-failover.html>
 - *⁸ <https://docs.couchbase.com/server/current/learn/clusters-and-availability/hard-failover.html>

第 12 章

運用

12.1 監視

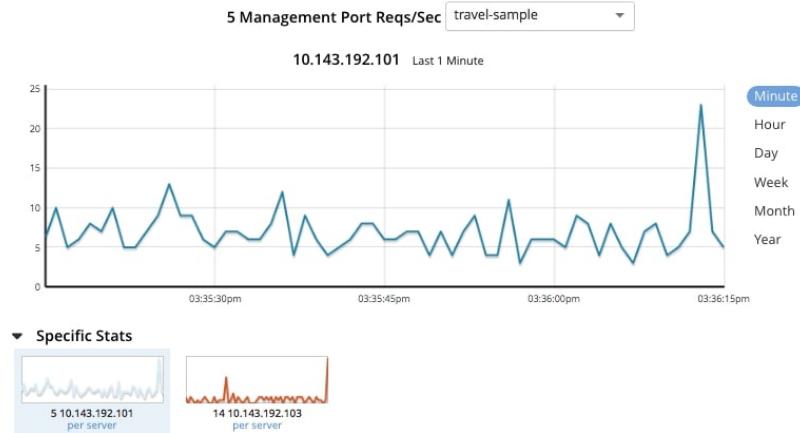
Couchbase Server における監視について見ていきます。

Couchbase Server の監視機能の対象は、CPU、メモリ利用量などノード(サーバー) レベルの監視と、データ量やアクセス状況などデータベースやクラスター レベルの監視の両方を含んでいます。

Web コンソール

Couchbase Server の Web コンソールは、様々な情報を期間と対象を変えながら表示することのできるダッシュボードを備えています。^{*1}

^{*1} <https://docs.couchbase.com/server/current/manage/monitor/ui-monitoring-statistics.html>



REST API

外部システムとの汎用的な連携をサポートするために、Couchbase Server は、REST API による監視情報へのアクセスを提供しています。Web コンソールで公開されている監視情報を REST API を使って取得することが可能です。^{*2}

コマンドラインツール

監視システムとの連携の手法として、(エージェント型監視ツールを典型として) CLI(コマンドラインインターフェイス)が必要とされるケースもあります。Couchbase Server は用途に応じた各種 CLI ツールを備えていますが、監視情報取得の用途として cbstats^{*3}があります。

^{*2} <https://docs.couchbase.com/server/current/rest-api/rest-statistics.html>

^{*3} <https://docs.couchbase.com/server/current/cli/cbstats-intro.html>

アラート E メール

Couchbase Server は、自動フェイルオーバー等の特定の状況が発生した際に、E メールを送付する機能を持っており、Web コンソール画面または CLI、REST API から設定することが可能です。^{*4}

Prometheus 連携

Prometheus^{*5}は、監視対象にアクセスしてデータを収集する Pull 型アーキテクチャに基づいています。Prometheus による監視対象サーバーは、Exporter と呼ばれるアクセスポイントを公開します。

Prometheus のサイト (Exporters and Integrations^{*6}) で、公開されている Exporter のリストを見ることができます。公開されている Exporter の 1 つとして、Couchbase Exporter^{*7}を利用することができます。

Couchbase Exporter を利用する際には、監視するクラスターを指定して、以下のようにプロセスを実行します。

```
./couchbase-exporter --couchbase.username Administrator --couchbase.ip
```

ブログ記事「Couchbase Monitoring Integration with Prometheus and Grafana」^{*8}にて、Couchbase Server と Prometheus および Grafana との連

^{*4} <https://docs.couchbase.com/server/current/manage/manage-settings/configure-alerts.html>

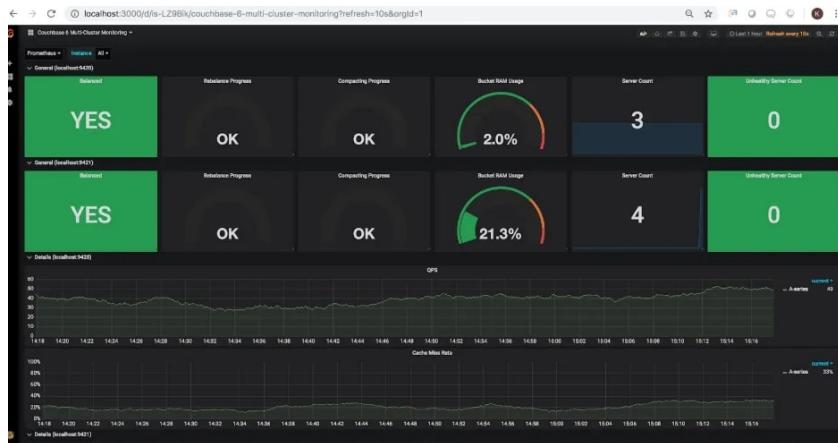
^{*5} <https://prometheus.io/>

^{*6} <https://prometheus.io/docs/instrumenting/exporters/>

^{*7} <https://github.com/couchbase/couchbase-exporter>

^{*8} <https://blog.couchbase.com/couchbase-monitoring-integration-with-prometheus-and-grafana/>

携に関する詳細な手順が説明されています。



(画像は、Couchbase Monitoring Integration with Prometheus and Grafana^{*9}より引用)

12.2 バックアップ

バックアップは、データベース管理において、欠かせないテーマです。

概要

Couchbase Server は、バックアップファイルの作成、バックアップファイルからの復元、バックアップファイルのアーカイブの機能を提供しています。これらの機能は、以下どちらの方法でも利用することができます。

^{*9} <https://blog.couchbase.com/couchbase-monitoring-integration-with-prometheus-and-grafana/>

- **Backup** サービス Web コンソールまたは REST API を用いて、スケジューリングを含めた操作が可能です。
- コマンドライン操作 `cbackupmgr` が提供されています。

■コラム: エディションによる差異

Backup サービスは、エンタープライズエディションでのみ利用可能です。

Backup サービス

本書では、コミュニティエディションに基づく記述を方針としているため、エンタープライズエディションでのみ利用可能な Backup サービスは対象外となります。

一方、本書で取り扱っていない他のサービスは、データ分析や全文検索など、ユースケースによって要否が決まり、必ずしも普遍的に必要な知識ではないのに対して、Backup サービスは、それらとは異なる性質を持ちます。

そこで、Backup サービスについては、ドキュメント参照先^{*10}を示すことで、機能の紹介に代えたいと思います。

コマンドラインツール

`cbackupmgr` コマンドを使ったバックアップから復元までの一連の操作については、コマンドライン操作に関する章で詳しく説明します。

ここでは、`cbackupmgr` の基本構文と（サブ）コマンドの一覧を示すため、`-h` オプションの出力を掲載します。

^{*10} <https://docs.couchbase.com/server/current/manage/manage-backup-and-restore/manage-backup-and-restore.html>

```
$ cbbbackupmgr -h
cbbbackupmgr [<command>] [<args>]

  backup      Backup a Couchbase cluster
  collect-logs Collect debugging information
  compact     Compact an incremental backup (Unsupported for 6.6.0)
  config      Create a new backup configuration
  help        Get extended help for a subcommand
  examine     Search inside a backup/repository for a specified doc
  info        Provide information about the archive
  merge       Merge incremental backups together (Enterprise Edition Only)
  remove      Delete a backup permanently
  restore     Restore an incremental backup

Optional Flags:

  --version          Prints version information
  -h,--help           Prints the help message
```

■コラム: エディションによる差異

上記の出力例中の「Enterprise Edition Only」という記載からも見て取れるように、cbbbackupmgrには、mergeやexamineサブコマンドのように、エンタープライズエディションでのみ利用できる機能があります。

第 13 章

コマンドライン操作

13.1 基本

Couchbase Server には、様々なコマンドラインツールが用意されています。ここでは、それらの中から主なものについて紹介します。すべてのツールに関する詳細な情報については、ドキュメント^{*1}を参照してください。

インストール場所

Couchbase Server をインストールした際に同時にインストールされるコマンドは、下記のパス以下 (ほとんどは bin 直下、一部 tools や、install ディレクトリの下) に存在します。

- **Linux:** /opt/couchbase/bin
- **Windows:** C:\Program Files\couchbase\server\bin
- **Mac OS X:** /Applications/Couchbase Server.app/Contents/Resources/couchbase-core/bin

本章の実行例では、コマンドへのパスが通っていることを前提としてい

^{*1} <https://docs.couchbase.com/server/current/cli/cli-intro.html>

ます。

13.2 クラスター管理: couchbase-cli

couchbase-cli は Couchbase Server の管理のため提供されている、中心となるコマンドラインインターフェイスです。

基本構文

基本的な構文は以下の通りです。

```
couchbase-cli <サブコマンド> [オプション]
```

サブコマンド

couchbase-cli で利用可能なサブコマンドを紹介します。

bucket-で始まる、バケット管理のための各種サブコマンドが用意されています。また、collection-manage サブコマンドを用いて、スコープやコレクションの作成、削除を行うことができます。

その他、クラスターの管理のための様々なサブコマンドが存在します。

これらの操作のほとんどは、Web コンソールを使って行うことができ、特に Couchbase Server を使い始めた人にとっては、そちらの方が直感的に利用することができるでしょう。一方、コマンドライン操作という選択肢が用意されていることは、本格的な運用にとって、操作の再現性などの面で重要です。

ここでは、可能な操作の紹介を意図しています。利用方法については、ド

キュメント^{*2}を参照してください。

^{*2} <https://docs.couchbase.com/server/current/cli/cbcli/couchbase-cli.html>

第13章 コマンドライン操作

13.2 クラスター管理: couchbase-cli

サブコマンド	内容
admin-role-manage	LDAP ユーザーロールを管理します。
bucket-compact	バケットのコンパクションを実行します。
bucket-create	新しいバケットを作成します。
bucket-delete	バケットを削除します。
bucket-edit	バケット設定を編集します。
bucket-flush	バケットのデータをフラッシュします。
bucket-list	すべてのバケットを一覧表示します。
cluster-edit	クラスター設定を編集します。
node-to-node-encryption	ノード間暗号化の有効化/無効化を設定します。
cluster-init	クラスターを初期化します。
ip-family	クラスターで使われている IP ファミリーを変更します。
collect-logs-start	ログ収集を開始します。
collect-logs-status	ログ収集ステータスを取得します。
collect-logs-stop	現在のログ収集タスクを停止します。
eventing-function-setup	Eventing サービス機能を管理します。
failover	クラスター内のサーバーをフェイルオーバーします。
group-manage	サーバーグループを管理します。
host-list	クラスター内のすべてのホストを一覧表示します。
node-init	ノードを初期化します。
rebalance	クラスター内のノード間でデータのリバランスを行います。
rebalance-status	現在のリバランスステータスを表示します。
rebalance-stop	現在のリバランスタスクを停止します。
recovery	以前にフェイルオーバーしたノードを回復します。
reset-admin-password	管理者パスワードをリセットします。
reset-cipher-suites	暗号スイートをデフォルトにリセットします。
server-add	サーバーをクラスターに追加します。
server-info	サーバーレベルの情報と統計を表示します。
server-list	クラスター内のすべてのサーバーを一覧表示します。
server-readd	フェイルオーバー後にサーバーをクラスターに追加し直します。
setting-alert	アラート設定を変更します。
setting-audit	監査ログの設定を変更します。
setting-autofailover	自動フェイルオーバー設定を変更します。
setting-autoreprovision	自動再プロビジョニングの設定を変更します。
setting-cluster	クラスター設定を変更します。
setting-compaction	圧縮設定を変更します。
setting-index	インデックス設定を変更します。
setting-ldap	LDAP 設定を変更します。
setting-notification	通知設定を変更します。
setting-password-policy	パスワードポリシーを変更します。
setting-query	クエリ設定を変更します。
setting-rebalance	リバランス設定を変更します。
setting-security	セキュリティポリシーを変更します。

13.3 クエリ: cbq

cbq を用いて、コマンドラインから、N1QL クエリを実行することができます。

実行例

--script オプションを用いて、下記のように、クエリを実行することができます。

```
$ cbq -u Administrator -p password -e "http://localhost:8091" \
--script="SELECT * FROM `travel-sample`.inventory.airline LIMIT 1;"
```

--script オプションを用いずに、下記のように、インタラクティブシェルモードで利用することも可能です。

```
$ cbq -u Administrator -p password -e "http://localhost:8091"
Connected to : http://localhost:8091/. Type Ctrl-D or \QUIT to exit.

Path to history file for the shell : /Users/<user name>/.cbq_history
cbq>SELECT * FROM `travel-sample`.inventory.airline LIMIT 1;
```

オプションや、シェルコマンドの詳細についてはドキュメント^{*3}を参照してください。

^{*3} <https://docs.couchbase.com/server/current/tools/cbq-shell.html>

13.4 データのインポート: cbimport

概要

`cbimport` を用いて、Couchbase Server へファイルからデータをインポートすることができます。入力に用いるファイルのフォーマットとして CSV と JSON をサポートしています。

ここでは、基本的な機能を紹介し、最後に、CSV ファイルからデータをインポートするケースについて、利用例を示して解説します。

詳細についてはドキュメント^{*4}を参照してください。

基本構文

```
cbimport [--version] [--help] <csv|json> [<args>]
```

```
cbimport csv [--cluster <url>] [--bucket <bucket_name>] [--dataset <path>]
[--username <username>] [--password <password>] [--generate-key <key_expr>]
[--limit-rows <num>] [--skip-rows <num>] [--field-separator <char>]
[--cacert <path>] [--no-ssl-verify] [--threads <num>] [--error-log <path>]
[--log-file <path>] [--verbose] [--field-delimiter <char>]
[--generator-delimiter <char>] [--ignore-fields <fields>]
[--scope-collection-exp <scope_collection_expression>]
```

^{*4} <https://docs.couchbase.com/server/current/tools/cbimport.html>

```
cbimport json [--cluster <url>] [--bucket <bucket_name>] [--dataset <path>]
               [--format <data_format>] [--username <username>] [--password <password>]
               [--generate-key <key_expr>] [--cacert <path>] [--no-ssl-verify]
               [--threads <num>] [--error-log <path>] [--log-file <path>] [--verbose]
               [--ignore-fields <fields>] [--field-delimiter <char>]
               [--generator-delimiter <char>]
               [--scope-collection-exp <scope_collection_expression>]
               [--bucket-quota <quota>]
```

ホストフォーマット

cbimport(および cbexport) コマンドに対して、ホスト名を指定する際には、以下のフォーマットが利用可能です。

- <addr>:<port>
- http://<addr>:<port>
- https://<addr>:<port>
- couchbase://<addr>:<port>
- couchbases://<addr>:<port>
- couchbase://<srv>
- couchbases://<srv>
- <addr>:<port>,<addr>:<port>
- <schema>:<addr>:<port>,<addr>:<port>

末尾に s のついた couchbases://は、(http://と https://の関係と同様)、セキュアな接続を意味します。

ホストフォーマットでは、複数のノードを指定したり、SRV レコードを用いることができます。詳細な解説や、有効な接続文字列の実例について、

ドキュメント^{*5}を参照してください。

JSON データセットフォーマット

json サブコマンドを用いて実行される場合、cbimport は、以下のフォーマットを持つファイルを入力として扱うことが可能です。

LINES

-format オプションとして、「lines」を指定します。

各行に、1つの JSON オブジェクトが記載されているファイルを入力とします。

以下に例を示します。

```
{"key": "mykey1", "value": "myvalue1"}  
{"key": "mykey2", "value": "myvalue2"}  
{"key": "mykey3", "value": "myvalue3"}  
{"key": "mykey4", "value": "myvalue4"}
```

LIST

-format オプションとして、「list」を指定します。

各要素が JSON ドキュメントである配列を含むファイルを入力とします。

ファイルには単一の配列のみを含めることができます。JSON データとして成立している限り、配列の要素がファイル上1行である必要はなく、複数行で指定できます。

以下に例を示します。

^{*5} <https://docs.couchbase.com/server/current/tools/cbimport-csv.html#host-formats>

```
[  
  {  
    "key": "mykey1",  
    "value": "myvalue1"  
  },  
  {"key": "mykey2", "value": "myvalue2"},  
  {"key": "mykey3", "value": "myvalue3"},  
  {"key": "mykey4", "value": "myvalue4"}  
]
```

CSV データセットフォーマット

csv サブコマンドを用いて実行される場合、cbimport は、CSV ファイルを入力とします。

JSON データに変換するために、フィールド名が必須になるため、CSV ファイルの 1 行目はデータではなく、フィールド（カラム）名である必要があります。

デフォルトでは、カンマ区切りが想定されていますが、`--field-separator` フラグを使用して、別の文字を区切り文字として扱うことが可能です。

ドキュメントキー

ファイルからデータをインポートする際、`--generate-key(-g)` オプションを利用して、ドキュメントのキーとして用いる項目を指定することができます。

指定方法は、`%user_id%` のように、フィールド名の前後を % で修飾します。

また、`#UUID#`というキーワードを用いて、キーを自動的に生成することもできます。キーは、`5565431f-9e0e-480d-abd6-c844a3ff6f9b` のよう

な UUID になります。

インポート先の指定

ファイルからデータをインポートする際、**-b** オプションにより、インポート先のバケット名を指定します。

また、ドキュメントを保存するスコープやコレクションを指定する**--scope-collection-exp** オプションがあります。このオプションでは、スコープとコレクションの指定に、「<スコープ名>.<コレクション名>」のようにドット表記を用います。入力データ中の特定のフィールド（カラム）の値と同じ名前のスコープやコレクションを保存先に指定することができ、その場合には、**--generate-key(-g)** オプション同様に、フィールド名の前後を % で修飾したものを用います。

データ型推論

--infer-types を指定すると、フィールド（カラム）の値からデータ型を推論して、対応する JSON のデータ型へ変換することができます。

利用例

下記のようなデータファイルがあるとします。

```
user_id,region,name
"001",japan,田中
"002",japan,佐藤
"011",us,Smith
```

以下のようなコマンドを用いて、このデータをインポートすることができます。

```
$ cbimport csv -c http://localhost:8091 -u Administrator -p password  
-d file:///user_data.csv -b user_management \  
-g %user_id% --scope-collection-exp %region%.users
```

-g(--generate-key) オプションとして、データに含まれる user_id が (%で修飾して) 指定されている他、--scope-collection-exp オプションで、スコープ名に、region が指定されているのが分かります。--infer-types を指定していないので、user_id は、文字列として扱われます。

13.5 データのエクスポート: cbexport

概要

cbexport を用いて、Couchbase Server から、データをファイルにエクスポートすることができます。出力されるファイルのフォーマットは JSON です。

ここでは、基本的な機能を紹介します。詳細についてはドキュメント^{*6}を参照してください。

基本構文

```
cbexport json [--cluster <url>] [--bucket <bucket_name>] [--format <data_format>]  
[--username <username>] [--password <password>] [--cacert <path>]  
[--no-ssl-verify] [--threads <num>] [--log-file <path>]  
[--include-key <key>] [--include-data <collection_string_list>]  
[--exclude-data <collection_string_list>] [--verbose]
```

^{*6} <https://docs.couchbase.com/server/current/tools/cbexport-json.html>

```
[--scope-field <scope_field>] [--collection-field <collection_field>]
```

データセットフォーマット

`cbexport` は、`cbimport` における JSON データの扱いと同様に、以下のフォーマットでファイルを出力することが可能です。

- **LINES** 各行が、1つの JSON オブジェクト
- **LIST** JSON オブジェクトを要素として持つ、1つの配列

13.6 バックアップ: cbbackupmgr

データベースにとって、バックアップは必須の機能です。

Couchbase Server のバックアップのために用意されている、`cbbackupmgr` コマンドの使用例を紹介します。

ここで紹介しきれない、使用可能なオプションや、全てのオプションの詳細については、ドキュメント^{*7}を参照してください。

本セクションの内容は、公式ドキュメントのチュートリアル^{*8}に基づいています。説明にあたって、Couchbase Server にサンプルデータとして準備されている `travel-sample` パケットと `beer-sample` パケットを利用します。

バックアップリポジトリの作成

`cbbackupmgr` の使用を開始するにあたって、まずバックアップを保存す

^{*7} <https://docs.couchbase.com/server/current/backup-restore/cbbackupmgr.html>

^{*8} <https://docs.couchbase.com/server/current/backup-restore/cbbackupmgr-tutorial.html>

るディレクトリを決定する必要があります。このディレクトリは、バックアップアーカイブと呼ばれます。

バックアップアーカイブには、バックアップリポジトリが含まれます。バックアップリポジトリは、configサブコマンドを使用して作成します。

このチュートリアルでは、/data/backupをバックアップアーカイブとして使用します。バックアップアーカイブには、空のディレクトリを選択します。cbbackupmgrコマンド初回実行時に、バックアップアーカイブとして構成されます。

バックアップリポジトリを構成する最も単純な方法は、クラスター全体をバックアップ対象とすることです。以下のように、バックアップ対象を指定するオプションを用いていない場合、クラスター全体がバックアップ対象となります。

ここでは、「cluster」という名前でバックアップリポジトリを作成します。

```
$ cbbackupmgr config -a /data/backup -r cluster  
Backup repository 'cluster' created successfully in archive '/data/ba
```

バックアップリポジトリ「cluster」がアーカイブ/data/backupに正常に作成されました

バックアップリポジトリを作成する際に、バックアップの対象を選択することができます。例えば、travel-sampleバケット内のインデックス定義のみをバックアップするとします。以下のように、`--include-data`オプションを用いて、バックアップ対象のバケットを指定することができます。また、`--disable-data`オプションにより、データのバックアップを無効化することにより、インデックス定義のみがバックアップされます。

ここでは「single」という名前でバックアップリポジトリを作成します。

```
$ cbbackupmgr config -a /data/backup -r single \
--include-data travel-sample --disable-data

Backup repository 'single' created successfully in archive '/data/bac
```

`config` サブコマンドは、データのバックアップ方法をカスタマイズするための多くのオプションを提供します。

バックアップ構成の確認

バックアップリポジトリを作成したので、バックアップアーカイブの情報を確認します。そのために、`info` サブコマンドを使用することができます。

```
$ cbbackupmgr info -a /data/backup --all
Name          | UUID                                     | Size   | # Repos
backup_repo   | 32c97d5f-821a-4284-840b-9ee7cf8733a3 | 0B    | 2
*  Name       | Size   | # Backups |
*  cluster    | 0B     | 0           |
```

`info` サブコマンドは、バックアップアーカイブ内のすべてのバックアップリポジトリとバックアップの情報を表示します。

ここでは、バックアップはまだ作成されていないので、バックアップリポジトリの情報のみが確認できます。

バックアップの作成

バックアップの作成を開始します。バックアップリポジトリには、バックアップの作成方法に関する情報が含まれているため、バックアップ作成時には、バックアップアーカイブの場所と、バックアップリポジトリ（の名前）、

そしてバックアップするターゲットクラスターの情報(認証情報を含む)を指定します。

「cluster」バックアップリポジトリで、ローカルホストで稼働しているクラスターのバックアップを作成します。

```
$ cbbackupmgr backup -c 127.0.0.1 -u Administrator -p password -a /data
Backing up to 2020-03-25T08_08_11.770436Z
Copied all data in 33.02s (Avg. 759.44KB/Sec) 38894 items / 24.47MB
beer-sample [=====] 100.00%
travel-sample [=====] 100.00%

Backup successfully completed
Backed up bucket "beer-sample" succeeded
Mutations backed up: 7303, Mutations failed to backup: 0
Deletions backed up: 0, Deletions failed to backup: 0
Backed up bucket "travel-sample" succeeded
Mutations backed up: 31591, Mutations failed to backup: 0
Deletions backed up: 0, Deletions failed to backup: 0
```

コマンドを実行すると、プログレスバーが出力されます。これは、バックアップの完了にかかる時間とデータ移動の速度を理解するのに役立ちます。バックアップの実行中、完了までの推定時間が示されます。すでにバックアップされているデータとアイテムの合計、および現在のデータ移動速度に関する情報も提供されます。

バックアップが正常に完了すると、「Backup successfully completed」というメッセージが表示され、バックアップされた変更(Mutations)と削除(Deletions)の内訳がバケットごとに表示されます。

「single」バックアップリポジトリでも、バックアップを実行して、2つのバックアップ実行の違いを確認してみましょう。

```
$ cbbackupmgr backup -a /data/backup -r single \
-c couchbase://127.0.0.1 -u Administrator -p password

Backing up to 2020-03-25T08_08_58.903046Z
Copied all data in 1s (Avg. 480B/Sec)          0 items / 480B
travel-sample [=====] 100.00%
Backup successfully completed
Backed up bucket "travel-sample" succeeded
Mutations backed up: 0, Mutations failed to backup: 0
Deletions backed up: 0, Deletions failed to backup: 0
```

「single」バックアップリポジトリは、`travel-sample` バケットからインデックス定義のみをバックアップするように構成されているため、`beer-sample` バケットの進行状況バーは表示されていません。また、バックアップするデータがはるかに少ないため、バックアップがより迅速に実行されたことがわかります。

これで、バックアップアーカイブにバックアップが作成されたので、`info` サブコマンドを使用してバックアップアーカイブの状態を確認してみましょう。

```
$ cbbackupmgr info -a /data/backup --all
Name      | UUID
backup_repo | 32c97d5f-821a-4284-840b-9ee7cf8733a3 | 54.35MB | 2
* Name      | Size      | # Backups |
* cluster   | 54.33MB  | 1           |
+
+   Backup            | Size      | Type    | Source
+   2020-03-25T08_08_11.770436Z | 54.33MB | FULL   | http://localhost:8091/_utils/backup/2020-03-25T08_08_11.770436Z
-
-   Bucket            | Size      | Items   | Mutations | Tombstones |
-   beer-sample       | 18.43MB  | 7303    | 7303     | 0           |
```

Bucket				
Name				
-	travel-sample	35.91MB	31591	31591 0
*	single	16.20KB	1	
Backup				
+	2020-03-25T08_08_58.903046Z	16.20KB	FULL	http://localhost:8080/_temp/cb-backupmgr/2020-03-25T08_08_58.903046Z/cluster/
Bucket				
-	travel-sample	16.20KB	0	0 0

いくつかのバックアップが作成されたので、`info` サブコマンドの出力内容が変わっています。「cluster」バックアップリポジトリには、バックアップが作成された時間に対応する名前のバックアップが1つ含まれていることがわかります。そのバックアップには2つのバケットが含まれており、バケット内のミューテーション、トゥームストーン、インデックスなどの数を確認できます。「single」バックアップリポジトリには1つのバックアップが含まれていますが、このバックアップには `travel-sample` バケットのみが含まれ、データ項目は0個で、10個のインデックスがあることがわかります。

なお、`info` コマンドはJSONフォーマットでの出力もサポートしています。

増分バックアップ

`cbackupmgr` の重要な特徴として、増分バックアップがあります。2回目以降のバックアップでは、バックアップリポジトリに既に存在しているデータは、バックアップの対象とはなりません。

増分バックアップの確認のため、一部のデータを変更した後、「cluster」バックアップリポジトリで `backup` サブコマンドを再度実行してみます。

```
$ cbbackupmgr backup -a /data/backup -r cluster -c couchbase://127.0.0.1:8091  
Backing up to 2020-03-25T08_08_11.770436Z  
Copied all data in 3s (Avg. 18.98KB/Sec) 4 items / 56.95KB  
travel-sample [=====] 100.00%  
beer-sample [=====] 100.00%  
  
Backup successfully completed  
Backed up bucket "beer-sample" succeeded  
Mutations backed up: 0, Mutations failed to backup: 0  
Deletions backed up: 0, Deletions failed to backup: 0  
Backed up bucket "travel-sample" succeeded  
Mutations backed up: 4, Mutations failed to backup: 0  
Deletions backed up: 0, Deletions failed to backup: 0
```

この実行例では、バックアップ実行前に 2 つのアイテムを更新して、2 つのアイテムを新規作成しています。バックアップされたのはそれらのデータのみであることが分かります。

`info` サブコマンドを使用して、バックアップアーカイブの情報を確認します。新しくバックアップが追加されており、そのバックアップの `Items` の数は 4 で、そのすべてが `Mutations` であることがわかります。これは、2 つの変更と 2 つの追加に対応しています。

```
$ cbbackupmgr info -a /data/backup --all  
Name | UUID | Size | # Backups |  
backup_repo | 32c97d5f-821a-4284-840b-9ee7cf8733a3 | 86.37MB | 2  
  
* Name | Size | # Backups |  
* cluster | 86.35MB | 2 |  
  
+ Backup | Size | Type | Source  
+ 2020-03-25T08_08_11.770436Z | 54.33MB | FULL | http://localhost:8091
```

```
-      Bucket      | Size      | Items   | Mutations | Tombstones |
-      beer-sample | 18.43MB  | 7303    | 7303     | 0          |
-
-      Bucket      | Size      | Items   | Mutations | Tombstones |
-      travel-sample | 35.91MB | 31591   | 31591   | 0          |
+
+  Backup           | Size      | Type    | Source
+  2020-03-25T08_41_21.461311Z | 32.02MB | INCR   | http://localhost:8091/_temp/cbbackupmgr/2020-03-25T08_41_21.461311Z/beer-sample
+
-      Bucket      | Size      | Items   | Mutations | Tombstones |
-      beer-sample | 16.00MB  | 0       | 0        | 0          |
-
-      Bucket      | Size      | Items   | Mutations | Tombstones |
-      travel-sample | 16.02MB | 4       | 4        | 0          |
*
*  Name      | Size      | # Backups |
*  single   | 16.20KB  | 1          |
+
+  Backup           | Size      | Type    | Source
+  2020-03-25T08_08_58.903046Z | 16.20KB | FULL   | http://localhost:8091/_temp/cbbackupmgr/2020-03-25T08_08_58.903046Z/travel-sample
-
-      Bucket      | Size      | Items   | Mutations | Tombstones |
-      travel-sample | 16.20KB | 0       | 0        | 0          |
```

バックアップの復元

バックアップデータができたので、そのバックアップデータをクラスターに復元してみましょう。データを復元するには、復元するバックアップの名前を知っている必要があります。名前を確認するには、`info` サブコマンドを使用して、バックアップアーカイブの内容を表示します。バックアップ名は常にタイムスタンプになります。

ここでは、「cluster」バックアップリポジトリの1回目のバックアップから復元します。

```
$ cbbackupmgr restore -a /data/backup -r cluster \
-c http://127.0.0.1:8091 -u Administrator -p password \
--start 2020-03-25T08_08_11.770436Z \
--end 2020-03-25T08_08_11.770436Z --force-updates

(1/1) Restoring backup 2020-03-25T08_08_11.770436Z
Copied all data in 2s (Avg. 19.96MB/Sec)      38894 items / 39.91MB
travel-sample          [=====] 100.00%
beer-sample            [=====] 100.00%

Restore completed successfully
```

復元を行う際には、`--start` フラグと`--end` フラグを使用して、復元するバックアップの範囲を指定します。ここでは、バックアップを1つだけ復元するため、`--start` と`--end` の両方に同じものを指定しています。

また、`--force-updates` フラグが利用されています。これにより、競合解決プロセスがスキップされ、クラスター上の値が新しく更新されている場合でも、バックアップされたデータで強制的に上書きします。ここで、2回目のバックアップ前に更新した2つの値を確認すると、最初のバックアップを作成したときの値に戻っていることがわかります。

`restore` サブコマンドでは、バックアップデータの一部を復元から除外するなど、さまざまなオプションを利用することができます。

バックアップのマージ

増分バックアップを行うと、バックアップを作成するたびにディスク容量が増加します。ディスクスペースは無限ではないため、スペースを再利用できるようにすることが重要です。

`merge` サブコマンドを使用して2つ以上のバックアップをマージすることができます。

「cluster」バックアップリポジトリには 2 つのバックアップがあるので、これらのバックアップをマージしてみます。

```
$ cbbackupmgr merge -a /data/backup -r cluster \
--start oldest --end latest

Merge completed successfully
```

■コラム: エディションによる差異

`merge` サブコマンドは、エンタープライズエディションでのみ利用できます。

バックアップのマージが成功したので、`list` サブコマンドを使用して、`merge` サブコマンドがバックアップアーカイブに与えた影響を確認します。

```
$ cbbackupmgr info -a /data/backup --all
Name      | UUID                                     | Size      | # Backups
backup_repo | 32c97d5f-821a-4284-840b-9ee7cf8733a3 | 54.35MB  | 2

* Name      | Size      | # Backups  |
* cluster   | 54.33MB   | 1          |

+ Backup           | Size      | Type       | Source
+ 2020-03-25T08_41_21.461311Z | 54.33MB  | MERGE - FULL | Merge

- Bucket          | Size      | Items     | Mutations | Tombstones
- beer-sample    | 18.43MB  | 7303     | 7303      | 0
```

```
-      Bucket      | Size      | Items   | Mutations | Tombstones
-      travel-sample | 35.91MB | 31593  | 31593     | 0
*
*   Name      | Size      | # Backups |
*   single    | 16.20KB  | 1        |
+
+   Backup          | Size      | Type    | Source
+   2020-03-25T08_08_58.903046Z | 16.20KB | FULL   | http://localhost:8080/_temp/cb-backupmgr/_backups/2020-03-25T08_08_58.903046Z
-
-      Bucket      | Size      | Items   | Mutations | Tombstones
-      travel-sample | 16.20KB | 0       | 0         | 0
```

「cluster」バックアップリポジトリに単一のバックアップがあることがわかります。このバックアップの名前は、最新のバックアップの名前を反映しています。

現在、`travel-sample` バケットのバックアップには、31593 個のデータ項目があります。マージ前の 2 番目のバックアップには、2 つの新しいアイテムが含まれていました。そのため、最初のバックアップのデータ項目数(31591 個)から 2 つ増えています。2 番目のバックアップに含まれていた残りの 2 つのアイテムは、新規アイテムではなく更新であるため、マージ中に重複が取り除かれ、マージ後のバックアップのアイテムの個数に影響を与えていません。

バックアップリポジトリの削除

バックアップリポジトリが不要になった場合は、`remove` サブコマンドを使用してバックアップリポジトリを削除できます。

以下は、「cluster」バックアップリポジトリを削除する例です。

```
$ cbbackupmgr remove -a /data/backup -r cluster  
Backup repository 'cluster' deleted successfully from archive '/data/
```

バックアップリポジトリ「cluster」がアーカイブ/data/backup から正常に削除されました。

ここで `list` サブコマンドを実行すると、「cluster」バックアップリポジトリが存在していないことを確認できます。

第 14 章

ユーティリティ

14.1 概要

Couchbase Server には、CLI リファレンス^{*1}に含まれているコマンドラインツールの他に、C SDK(Couchbase C Client) の一部として `libcouchbase` コマンドにカテゴライズされるユーティリティが利用可能です。^{*2}

インストール

ここで紹介するユーティリティは、Couchbase Server をインストールする際に、他のコマンドラインツール同様、インストールされます。

また、以下の方法で、Couchbase Server がインストールされていない環境に、C SDK をインストールし、これらのユーティリティを用いて、リモートのクラスターに対して、操作を行うことができます。

CentOS/RHEL では、以下のようにインストールします。

^{*1} <https://docs.couchbase.com/server/current/cli/cli-intro.html>

^{*2} <https://docs.couchbase.com/c-sdk/current/hello-world/start-using-sdk.html>

```
$ sudo yum check-update  
$ sudo yum search libcouchbase  
$ sudo yum install libcouchbase3 libcouchbase-devel libcouchbase3-tools
```

Mac では、以下のようにインストールします。

```
$ brew update  
$ brew install libcouchbase
```

Windows については、バイナリをダウンロードして利用します。詳細は、ドキュメント^{*3}を参照してください。

その他の OS については、リリースノート^{*4}を参照してください。

共通オプション

`libcouchbase` コマンドの共通オプションとして、以下があります。

^{*3} <https://docs.couchbase.com/c-sdk/current/project-docs/sdk-release-notes.html#install-windows>

^{*4} <https://docs.couchbase.com/c-sdk/current/project-docs/sdk-release-notes.html>

オプション	省略形	説明
--spec	-U	接続文字列 [デフォルト: couchbase://localhost/default]
--username	-u	ユーザー名
--password	-P	パスワード
--timings	-T	終了時に実行されたコマンドの待ち時間を示すヒストグラムを表示
--verbose	-v	デバッグ出力を有効にする
--cparam	-D	追加のクライアントオプション指定
--compress	-y	送信データの圧縮をオンにする
--truststorepath	なし	トラストストアへのパス
--certpath	なし	サーバー SSL 証明書へのパス
--keypath	なし	クライアント SSL 密密鍵へのパス
--dump	なし	操作完了後に詳細な内部状態をダンプする

クライアントオプション

追加のクライアントオプションを、キーと値のペアとして、D スイッチ (-D) と共に指定することができます。下記に記述例を示します。

```
-Doption1=value1 -Doption2=value2
```

あるいは、以下のように、接続文字列に含めて利用することができます。

```
couchbase://host/bucket?option1=value1&option2=value2
```

クライアントオプションには、以下のものがあります。

- `operation_timeout=SECONDS`: タイムアウト秒数を指定します。これは、クライアントが操作の完了を待つ時間です。デフォルトは 2.5 秒です。
- `config_cache=PATH`: クライアントからクラスターへの接続時に、ファイルベースの構成キャッシュを利用できるようにします。ファイルが存在しない場合、クライアントがクラスターに接続した際に、

ブートストラップ情報をファイルにキャッシュします。

- `certpath=PATH`: サーバー SSL 証明書へのパス (`couchbases://`スキームでのみ適用可能)。
- `ssl=no_verify`: SSL 検証を一時的に無効にします (`couchbases://`スキームでのみ適用可能)。
- `sasl_mech_force=MECHANISM`: 初期接続時に、特定の SASL メカニズムを強制的に使用します。現在サポートされているメカニズムは、`PLAIN` と `PCRAM-MD5` です。

14.2 ストレステストツール

概要

データベースを運用するに当たって、ベンチマーク（ストレス）テストは重要な要素です。

Couchbase Server では、Data サービスと Query サービスという 2 種類のサービスにより、データアクセス方法を提供していますが、それぞれのサービスに対して、下記のように異なるツールが用意されています。

サービス	ツール
Data サービス	<code>pillowfight</code>
Query サービス	<code>n1qlback</code>

pillowfight

`pillowfight`⁵を使って、Data サービスに対する、ドキュメント操作のストレステストを実施することができます。

⁵ https://docs.couchbase.com/sdk-api/couchbase-c-client-3.2.0/md_doc_cbc-pillowfight.html

`pillowfight` のオプションを示します。

オプション	省略形	説明
--batch-size	-B	バッチ処理の数 [デフォルト: 100]
--num-items	-I	アイテムの数 [デフォルト: 1000]
--key-prefix	-p	キー接頭辞の指定
--num-threads	-t	スレッドの数 [デフォルト: 1]
--random-body	-R	ドキュメント本文をランダム化する [デフォルト: FALSE]
--set-pct	-r	ミューテーション操作の割合 [デフォルト: 33]
--no-population	-n	ポピュレーション（入力）をスキップする [デフォルト: FALSE]
--min-size	-m	最小データ（ドキュメント）サイズ [デフォルト: 50]
--max-size	-M	最大データ（ドキュメント）サイズ [デフォルト: 5120]
--pause-at-end	-E	実行終了時に一時停止する（ユーザー入力まで接続を保持する）[デフォルト: FALSE]
--num-cycles	-c	終了するまで実行されるサイクル数。-1に設定すると、無限にループする [デフォルト: -1]
--sequential	なし	（ランダムアクセスでなく）シーケンシャルアクセスを使用 [デフォルト: FALSE]
--start-at	なし	シーケンシャルアクセスの場合に、最初の値を設定 [デフォルト: 0]
--docs	なし	ロードするユーザードキュメント（--min-size と--max-size を上書き）
--json	-J	（bytesではなく）JSONデータの書き込みを有効にする [デフォルト: FALSE]
--subdoc	なし	fulldoc 操作の代わりに subdoc 操作を使用 [デフォルト: FALSE]
--noop	なし	ドキュメント操作の代わりに、NOOP オペレーション（軽量ワークロード）を使用
--pathcount	なし	コマンドあたりのサブドキュメントパスの数 [デフォルト: 1]
--populate-only	なし	ドキュメントのポピュレーション（入力）のみを実施し、終了する [デフォルト: FALSE]
--expiry	-e	アイテムの TTL を設定 [デフォルト: 0]
--collection	なし	コレクションの（スコープを含む）フルパス（複数指定可能）
--durability	-d	耐久レベル [デフォルト: none]
--persist-to	なし	アイテムが指定した数のノードに永続化されるまでブロックする（"-1"の場合、すばやく失敗）
--replicate-to	なし	アイテムが指定した数のノードに複製されるまでブロックする（"-1"の場合、すぐ失敗）
--lock	なし	更新用ロックキーへの時間設定（ゼロの場合、ロックされない） [デフォルト: 0]
--help	-?	ヘルプメッセージを表示する

`pillowfight` の実行例を示します。

1スレッド当たり1バッチ処理を20スレッドで実行

```
$ cbc-pillowfight -t 20 -B 1 -u Administrator -P password
```

50 アイテムのデータセットを使った 100 回のイテレーション

```
$ cbc-pillowfight -c 100 -I 50 -u Administrator -P password
```

サイズ 100k の JSON ドキュメントを利用

```
$ cbc-pillowfight --json -m 100000 -M 100000 -u Administrator -P password
```

サブドキュメント操作によるストレステスト

```
$ cbc-pillowfight --json --subdoc --set-pct 100 -u Administrator -P password
```

JSON ドキュメントの有効期限を 10 秒に設定

```
$ cbc-pillowfight -U couchbase://localhost/pillow --json -e 10 -u Administrator -P password
```

最後の例では、接続先を明示的に指定しています。その他の例では、デフォルトの接続先 (couchbase://localhost/default) が利用されています。

n1qlback

`n1qlback`^{*6}を使って、Query サービスに対する、クエリ操作のストレステストを実施することができます。

`n1qlback`は、指定された数のスレッドを作成し、それぞれがユーザー定義のクエリのセットを実行します。

`n1qlback`では、実行するクエリを含むファイルへのパスを渡します。このファイルでは、下記のような JSON 形式でクエリリクエストを表現し、1 行に 1 リクエストを記載します。

単純なクエリの場合、`statement` フィールドのみを設定します。

```
{"statement":"SELECT country FROM 'travel-sample'"}  
{"statement":"SELECT country, COUNT(country) FROM 'travel-sample' GROUP BY country"}
```

次の例は、名前付きパラメータの使用方法を示しています。

```
{"statement":"SELECT RAW META().id FROM 'travel-sample' WHERE type=$type LIMIT 1", "$type": "airline"}
```

`n1qlback` のオプションを示します。

- `-f, --queryfile=PATH`: 実行するクエリを JSON 形式で記述したファイルへのパスを指定します。
- `-t, --num-threads=NTHREADS`: 同時に実行するスレッド数 (クライアントインスタンスの数) を設定します。

^{*6} https://docs.couchbase.com/sdk-api/couchbase-c-client-3.2.0/mb_doc_cbc-n1qlback.html

- `-e, --error-log=PATH`: エラーログを出力するファイルへのパスを指定します。失敗したクエリとエラーの詳細がファイルへ書き込まれます。

最後に、`n1qlback` の実行例を示します。

以下の例では、クエリリクエストを記載したファイル `queries.txt` を指定し、5つのスレッドで実行しています。

```
$ cbc-n1qlback -U couchbase://192.168.72.101/a_bucket -t 5 -f queries
```

14.3 その他のツール

`pillowfight` と `n1qlback` 以外に、以下のツールがあります。

ここでは紹介に留めますが、詳細は、各ツールの API Doc を参照してください。

- `cbc`^{*7}: クラスターと通信するためのユーティリティ
- `cbc-subdoc`^{*8}: サブドキュメント API に由来するコマンドを用いてデータにアクセスすることのできるインタラクティブシェルを実行するためのユーティリティ
- `cbcrc`^{*9}: `cbc` と `cbc-pillowfight` のためのデフォルト値を定義するために利用される、オプションの構成ファイル

^{*7} https://docs.couchbase.com/sdk-api/couchbase-c-client-3.2.1/md_doc_cbc.html

^{*8} https://docs.couchbase.com/sdk-api/couchbase-c-client-3.2.1/md_doc_cbc-subdoc.html

^{*9} https://docs.couchbase.com/sdk-api/couchbase-c-client-3.2.1/md_doc_cbcrc.html

第 15 章

環境設定

本章は、Couchbase Server の環境設定に関する網羅的な解説、あるいは実際に設定を行うために必要な手順の説明を意図していません。

アプリケーションの要件や、システム環境に応じて、意思決定が必要な設定の紹介を通じて、Couchbase Server に特有の概念や機能に関して理解を深めていただくことを目的としています。

網羅的な設定項目の解説については、適宜ドキュメント^{*1}を参照してください。

15.1 メモリからのデータ排出 (Ejection)

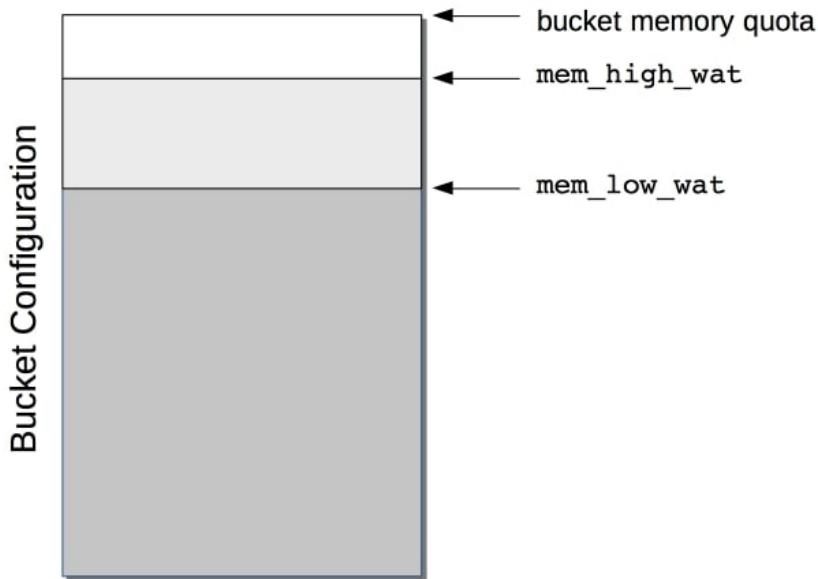
概要

Couchbase Server のバケットには、メモリクオータが設定され、専用のメモリ領域が確保されます。Couchbase Server は、このメモリ領域を用いて、可能な限りすべてのデータをメモリ上に保持しようとします。

メモリ内のデータがバケットのメモリクオータの 85 % (ハイウォーター

^{*1} <https://docs.couchbase.com/server/current/manage/manage-settings/manage-settings.html>

マークと呼ばれます)に達した場合、データがメモリから排出(削除)され始めます。この際、最近使用されていないデータから優先的に排出されます。この排出プロセスは、メモリクォータの75%（ローウォーターマークと呼ばれます）を下回るまで続きます。



(画像は、Couchbase Server ドキュメント Ejection^{*2}より引用)

^{*2} <https://docs.couchbase.com/server/current/learn/buckets-memory-and-storage/memory.html#ejection>

排出方法オプション

メモリからのデータ排出時のオプションとして、次のいずれかの方法を選択します。^{*3}

- **Value-only** 値のみが排出され、キーとメタデータはメモリに残ります。より多くのメモリが必要ですが、最高の性能が提供されます。
- **Full** (キー、メタデータ、値を含む) すべてが排出されます。メモリ要件を軽減します。

選択基準

排出方法の選択基準として、以下のように整理することができます。

- **ドキュメント数が一定または予測可能** 事前の計画に基づいたリソース設計により、**Value-only** を選択することができ、最高の性能を得ることができます。
- **ドキュメント数の増加ペースの予測が難しい** キーやメタデータは、それ自体のサイズは決して大きくありませんが、ドキュメント数に応じて増えるため、全てがメモリに収まりきれなくなる事態が起こりえます。この場合、**Full** を選択することで、メモリ容量超過の問題を回避することができます。

^{*3} ブログ記事「A Tale of Two Ejection Methods: Value-only vs. Full」(<https://blog.couchbase.com/a-tale-of-two-ejection-methods-value-only-vs-full/>) にて、オプションの選択が、性能とメモリ消費に与える影響が詳細に解説されています。

設定変更時の注意点

排出方法の設定は、バケット作成時に行えるだけではなく、作成後の変更も可能です。ただし、この変更の反映には、(設定変更時に暗黙に行われる)バケットのリストアートが伴うことに注意が必要です。そのため、アプリケーション実行中に設定変更を行った場合、アプリケーションからの接続がリセットされる事態が発生する可能性があります。

メモリ消費状況の把握

上述のように、排出方法の選択は、クラスターの性能と大きな関係があります。その意味で、安全性を重視して安易に **Full** を選択するのではなく、**Value-only** を選択して、高い性能を実現したいと考えるのは自然です。

また、事前の計画に基づいて、**Value-only** を選択した場合であっても、計画外の事態が発生する可能性を考慮することは依然として重要です。

ここでは、上記のような課題に対して利用できる機能を紹介します。

監視

Web コンソールでは、下記のように、RAM に占めるデータおよびメタデータに関する情報を確認することができます。

ここでは、機能を紹介するためにダッシュボード画面を示しましたが、Couchbase Server では、ダッシュボードと同内容の監視項目をコマンドラインや、REST API を用いて利用することができるため、これらを外部監視システムとの連携のために利用することが考えられます。



**3.04MB user data in RAM
per server**



**665KB metadata in RAM
per server**

警告

また、Couchbase Server には、特定の監視項目について警告を発生させる機能があり、メタデータオーバーヘッドに関する警告を利用するることができます。

下のような、Web コンソール上のポップアップの他、E メールを送信することができます。



[17 Jul, 2020, 1:58:01 AM] - Metadata overhead warning. Over 52% of RAM allocated to bucket "test" on node "127.0.0.1" is taken up by keys and metadata.

15.2 コンパクション

概要

Couchbase Server におけるデータの挿入、更新、削除は全て、物理的にディスク上のデータファイルに対する「追加」書き込みとして実現されます。その中でも、削除されたドキュメントの情報は、特に「トゥームストーン (Tombstone)」と呼ばれます。

この結果として生じる、物理データファイルと論理データとの (特にデータが削除された場合に顕著な) ギャップは、コンパクションと呼ばれるプロセスを通じて、定期的に解消されます。

コンパクションは、Couchbase Server の実行中にバックグラウンドプロセスとして実行されます。データベース操作をシャットダウンまたは一時停止する必要はありません。

コンパクションが実行される際には、新しいファイルが作成されます。コンパクション実行中、既存のデータファイルはそのまま残り、情報の格納やインデックスデータの更新に引き続き使用されます。このプロセスにより、コンパクションが行われている間もデータベースが引き続き使用可能にな

ります。コンパクションが完了すると、古いデータファイルが無効になります。同時に、新しく作成されたデータファイルを使った更新が開始されます。その後、古いデータファイルが削除されます。

Couchbase Server ではコンパクションは自動的に実行されます。

また、REST API^{*4}や、コマンドラインツール (`couchbase-cli`) を用いて、コンパクションを手動で実行することも可能です。

運用上の留意点

コンパクションの運用にあたり、下記に注意を払う必要があります。

- **余裕のある CPU 利用状況** コンパクションプロセスは CPU を集中的に使用します。
- **十分なディスク領域** コンパクションは新しいファイルを作成します。そのため、コンパクションの実行には、現在のデータファイルの倍のディスク領域が必要になります。

Couchbase Server では、コンパクションが実行される前に、ディスク領域がチェックされます。使用可能なディスクスペースが現在のデータファイルサイズの 2 倍未満の場合、コンパクション処理は実行されずにログに警告が出力されます。

オートコンパクション

オートコンパクション実行のトリガーとして、下記の 2 つがあります。

- **データファイルの断片化** データファイルが断片化されている割合をパーセンテージで設定します。例えば、10% に設定されている場合、データファイルが 10% 断片化された時点でオートコンパクションが

^{*4} <https://docs.couchbase.com/server/current/rest-api/rest-compact-post.html>

実行されます。

- **時間帯** オートコンパクションを許可する時間帯を設定して、それ以外の時間帯にオートコンパクションが行われないようにすることができます。デフォルトでは、コンパクションは順次実行されます。

15.3 代替アドレス

概要

Couchbase Server では、クラスターを構成するノードに対して、プリンシパルアドレスに加えて、代替 (Alternate) アドレス^{*5}を割り当てることができます（また、代替ポート番号をそのノードで実行されているサービスに割り当てることもできます）。

プリンシパルアドレスは、クラスターの初期構成時に設定されるアドレスです。代替アドレスは必要に応じて追加します。

代替アドレスの設定は CLI^{*6}または REST API^{*7}で実行可能です。

ここでは、代替アドレスが使われる 2 つのパターンを紹介します。

Couchbase Server の特殊性との関係

まず、代替アドレス設定について理解するための前提を整理します。

1 つのサーバーに複数のアドレスが割り当てられることは珍しいことではありませんが、分散アーキテクチャーを持つクラスターにおいては、モノリシックなアーキテクチャであれば考慮する必要がない面を、考慮する必要が

^{*5} <https://docs.couchbase.com/server/current/learn/clusters-and-availability/connectivity.html#alternate-addresses>

^{*6} <https://docs.couchbase.com/server/current/cli/cbcli/couchbase-cli-setting-alternate-address.html>

^{*7} <https://docs.couchbase.com/server/current/rest-api/rest-set-up-alternate-address.html>

あります。

Couchbase Server では、クライアント/アプリケーションがクラスターに接続する際には、クラスターマップという要素が介在します。初期接続は、特定のノード（複数指定可能）に対して行われますが、接続確立（ブートストラップ）時に、接続したノードからクラスターマップを受け取り、それ以降のリクエストは、対象となるノードに対して直接行われます。

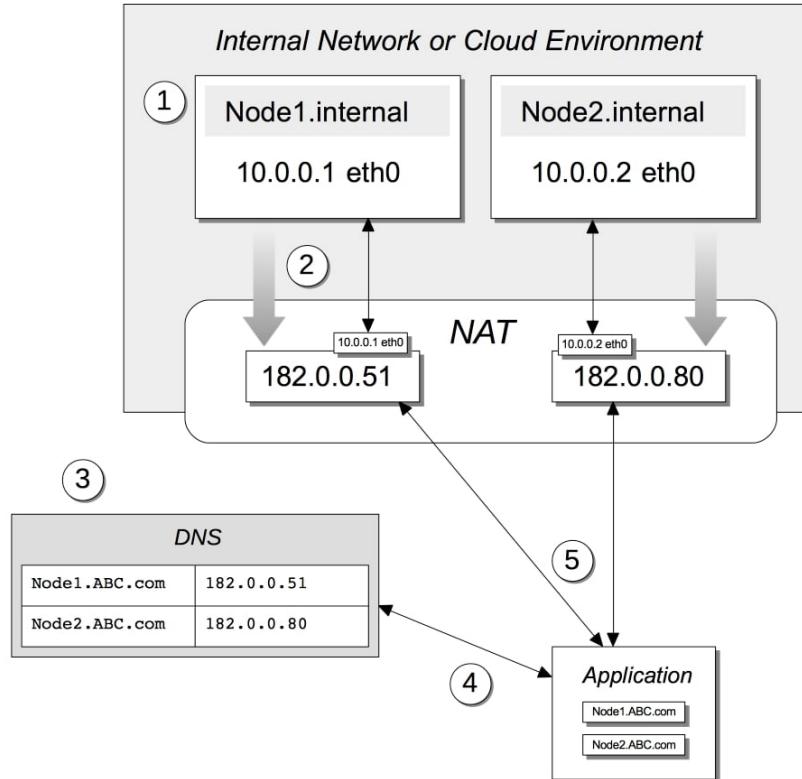
そのため、ブートストラップのためにクライアント/アプリケーションが明示的に利用している特定のアクセス先へのリクエストだけを考えて、IP やドメイン名の解決が外部的に整合性が取れていれば良い訳ではなく、クラスターを構成する全てのノードに関する内部情報（クラスターマップ）が代替アドレスと整合している必要があります。

外部ネットワークからの接続

外部ネットワークからクラスターへ接続するケースについて、下記に図示します。

まず注意しておくと、この図ではアプリケーションが外部ネットワークにあり、内部ネットワークにある Couchbase Server を利用していますが、これは必ずしも典型的なケースとは言えません。一般的には、アプリケーションと Couchbase Server は同じ内部ネットワークにあり、その場合、アプリケーションと Couchbase Server との接続には、プライベート IP が使われれば事足ります。この場合、代替アドレスを設定する必要はありません。

外部ネットワークにあるアプリケーションとの接続に限らず、複数の異なるネットワークにあるクラスター間で XDCR を実行する場合も、このケースに該当します。



(画像は、Couchbase Server ドキュメント Alternate Addresses - Internal Network or Cloud Access^{*8}より引用)

このケースでは、外部からの参照のために、ネットワークアドレス変換(NAT)機能で、各サーバーにパブリックIPを付与し、それらを代替アド

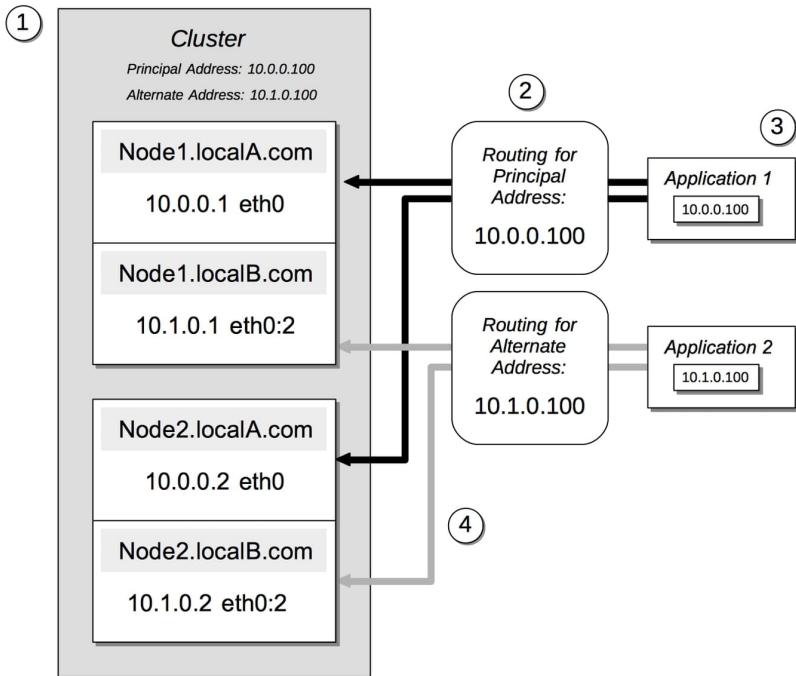
^{*8} <https://docs.couchbase.com/server/current/learn/clusters-and-availability/connectivity.html#internal-network-or-cloud-access>

レスに割り当てます。

プリンシパルアドレスには、プライベート IP アドレスが利用されます。ここではクライアントからの接続に焦点を当てていますが、プリンシパルアドレスは、クライアントからの接続のみではなく、クラスター内のノード間の接続にも利用されることには、理解しておく価値があります。このことから、プリンシパルアドレスに、単に外部アドレスを割り当てるとは適切ではない、ということがわかります。

デュアルネットワーク

次の図で示すように、セキュリティを最適化するために、デュアルネットワークを使用して複数のアプリケーションのノードへのアクセスを分離することが考えられます。



(画像は、Couchbase Server ドキュメント Alternate Addresses - Dual Network^{*9}より引用)

ここでは、2つのサーバーがあり、それぞれプライマリアドレスおよびセカンダリアドレスを持っています。それぞれのアドレスを、プリンシパルアドレスと代替アドレスに割り当てています。そして、2つのアプリケーションは、それぞれ割り当てられたプリンシパルアドレス、または代替アドレス

^{*9} <https://docs.couchbase.com/server/current/learn/clusters-and-availability/connectivity.html#dual-network>

のいずれかを利用します。

第 16 章

SDK と外部システム連携

16.1 SDK

サポートされるプログラミング言語

Couchbase Server には、各種プログラミング言語用の SDK が提供されています。

以下のプログラミング言語が、公式サポートされています。

- Java^{*1}
- Scala^{*2}
- .Net^{*3}
- C^{*4}
- Node.js^{*5}
- PHP^{*6}

^{*1} <https://docs.couchbase.com/java-sdk/current/hello-world/overview.html>

^{*2} <https://docs.couchbase.com/scala-sdk/current/hello-world/overview.html>

^{*3} <https://docs.couchbase.com/dotnet-sdk/current/hello-world/overview.html>

^{*4} <https://docs.couchbase.com/c-sdk/current/hello-world/overview.html>

^{*5} <https://docs.couchbase.com/nodejs-sdk/current/hello-world/overview.html>

^{*6} <https://docs.couchbase.com/php-sdk/current/hello-world/overview.html>

- Python^{*7}
- Ruby^{*8}
- Go^{*9}

また、デベロッパープレビューとして、Kotlin SDK^{*10}があります。

さらに、コミュニティサポートとして Rust SDK^{*11}があります。

拡張ライブラリ

マルチドキュメント(分散)トランザクションのためのライブラリが、以下の言語向けに提供されています。

- Java^{*12}
- .Net^{*13}
- C++^{*14}

16.2 外部システム連携

コネクター

Couchbase Serverと外部システムを連携するために利用できる、以下のコネクターが提供されています。

^{*7} <https://docs.couchbase.com/python-sdk/current/hello-world/overview.html>

^{*8} <https://docs.couchbase.com/ruby-sdk/current/hello-world/overview.html>

^{*9} <https://docs.couchbase.com/go-sdk/current/hello-world/overview.html>

^{*10} <https://docs.couchbase.com/kotlin-sdk/current/hello-world/overview.html>

^{*11} <https://github.com/couchbaselabs/couchbase-rs>

^{*12} <https://docs.couchbase.com/java-sdk/current/howtos/distributed-acid-transactions-from-the-sdk.html>

^{*13} <https://docs.couchbase.com/dotnet-sdk/current/howtos/distributed-acid-transactions-from-the-sdk.html>

^{*14} <https://docs.couchbase.com/cxx-txns/current/distributed-acid-transactions-from-the-sdk.html>

- Elasticsearch
- Kafka
- Spark

詳細は、ドキュメント (Connector Guides^{*15}) を参照してください。

^{*15} <https://docs.couchbase.com/server/current/connectors/intro.html>

第 17 章

Node.js アプリケーション 開発

17.1 なぜ、Node.js と Couchbase Server の組み合わせなのか

JSON データは、JavaScript と高い親和性を持っています。Couchbase Server を用いることによって、JSON データとクエリ言語の両方の利点を活用することができます。

JSON データの利点

まず、アプリケーションのデータ層に、リレーションナルデータベースを用いる場合に、問題となり得る点を以下に挙げます。

- 第一正規形テーブル構造と、アプリケーションが扱うデータモデル（ドメインオブジェクト）との間にギャップがあり得るため、データ層とアプリケーションの間で、データ構造を変換する必要が生じる場合がある。

- 厳格なテーブルスキーマに基づくデータ管理は、アプリケーション設計・開発の柔軟さを損なう場合がある。

これに対して、データ層が JSON データを許容した場合、以下の利点を得ることができます。

- データは、第一正規形であることが要請されないため、アプリケーションが扱うデータモデル（ドメインオブジェクト）をそのまま格納することができる。
- アプリケーション設計・開発において、データモデルの変化に柔軟に対応できる。

クエリ言語の利点

標準化されたクエリ言語である SQL が様々な異なるデータベースで利用できることは、リレーショナルデータベースの繁栄の大きな要因であり、SQL の習得は、アプリケーション開発者にとって、必修とさえ言えるものとなっていました。

Couchbase Server を選択することで、開発者は、既存の SQL の知識・経験を活用することができます。

17.2 キーバリュー操作

Couchbase Server では、開発者は、クエリを利用するだけでなく、キー指定によるデータ（ドキュメント）の操作を行うことが可能です。これをキーバリュー操作（Key Value Operations）と呼びます。

ここでは、アプリケーション開発におけるキーバリュー操作について、プログラミング例を用いて紹介します。詳細についてはドキュメント^{*1}を参照

^{*1} <https://docs.couchbase.com/nodejs-sdk/current/howtos/kv-operations.html>

してください。

CRUD 操作

キーバリュー操作の基本となる CRUD 操作に関して、以下のようなメソッドが利用可能です。

生成 (CREATE)

```
const result = await collection.insert(key, document);
```

読み取り (READ)

```
const result = await collection.get(key);
document = result.value;
```

更新 (UPDATE)

```
let result = await collection.replace(key, document);
```

削除 (DELETE)

```
const result = await collection.remove(key);
```

生成または更新 (UPsert)

```
let result = await collection.upsert(key, document);
```

UPsert メソッドは、同一のキーを持つデータが存在しなければ、生成 (CREATE/INSERT) を行い、既に同一のキーを持つデータが存在している場合は更新 (UPDATE/REPLACE) を行います。

サブドキュメント操作

データを参照する際に、ドキュメント全体ではなく、一部を参照すれば十分な場合があります。また、データの更新を行う場合、ドキュメントを取得し、必要な箇所を編集した上で、同じキーで保存するのは単純で分かりやすい操作ですが、ドキュメントの指定箇所の更新を指示することができれば十分な場合があります。このような操作は、N1QL を利用しても行うことができますが、キーバリュー操作でも実行することができます。これをサブドキュメント操作と呼びます。

サブドキュメント操作の基本的な種類を、プログラミング例を用いて紹介します。

生成 (CREATE)

```
await collection.mutateIn(key, [
  couchbase.MutateInSpec.insert("purchases.complete", [42, true, "None"]),
]);
```

読み取り (READ)

```
var result = await collection.lookupIn(key, [
  couchbase.LookupInSpec.get("addresses.delivery.country"),
]);
var country = result.content[0].value;
```

更新 (UPDATE)

```
await collection.mutateIn(key, [
  couchbase.MutateInSpec.replace("email", "xyz@example.com"),
]);
```

削除 (DELETE)

```
await collection.mutateIn(key, [
  couchbase.MutateInSpec.remove("addresses.billing"),
]);
```

生成または更新 (UPsert)

```
await collection.mutateIn(key, [
  couchbase.MutateInSpec.upsert("fax", "123-456-7890"),
]);
```

UPSERT メソッドは、指定したフィールドが存在しなければ追加し、既にフィールドが存在している場合はその値を更新します。

存在確認

```
var result = await collection.lookupIn(key, [
  couchbase.LookupInSpec.exists("purchases.pending[-1]"),
]);
console.log("Path exists? ", result.content[0].value);
```

データを取得するのではなく、指定したパスが存在するかどうかを確認することができます。

複数操作の同時リクエスト

上掲のコードを見れば分かりますが、サブドキュメント操作は、配列型の引数で与えられています。これまでの例では、要素が1つの配列が使われていますが、下記のように複数の操作を一度に与えることが可能です。

```
var result = await collection.lookupIn(key, [
  couchbase.LookupInSpec.get("addresses.delivery.country"),
  couchbase.LookupInSpec.exists("purchases.pending[-1]"),
]);
```

```
console.log(result.content[0].value);
console.log("Path exists?", result.content[1].value);
```

その他の操作

ここでは、CRUD 操作を中心に紹介しましたが、Couchbase Server のサブドキュメント操作の範囲は上記に留まるものではありません。

例えば、JSON ドキュメントは値を配列として持つことができますが、配列に対する値の追加・挿入を実行することができます。

また、値を数値として持つフィールドに対して、一旦値を取得した後に、増減した値で更新するのではなく、直接値の変更幅を指定するカウンター操作も用意されています。

その他、詳細についてはドキュメント^{*2}を参照してください。

■コラム：バルク操作 API

Couchbase Server のキーバリュー操作について見てきましたが、読者の中には、サブドキュメント操作については、複数の操作を一括して（バルクで）リクエストする API について解説しているのに対して、ドキュメントに関する CRUD 操作では、そのようなバルク操作について触れられていない点に注目された人がいるかもしれません。

この違いは、意図的な説明の省略によるものではなく、Couchbase Server の SDK に由来するものです。

Couchbase Server SDK の API には、同期型と非同期型の 2 種類が提供されています。そして、同期型の API の内部では、非同期型 API

^{*2} <https://docs.couchbase.com/nodejs-sdk/current/howtos/subdocument-operations.html>

が用いられ、結果が戻される前に、内部的に処理がブロック（同期）されることで、非同期処理が隠されています。非同期プログラミングについては、同期的なプログラミングとは異なる知識が必要であり、十分に理解せずに用いることには弊害が伴うため、同期型のラッパー API が提供されているのは理由があることだと言えます。

ここに、バルク処理の API を付け加えることは、本来の処理を、二重に隠すことにつながります。非同期 API の恩恵は、前後の処理を含めて可能な限り非同期で実行することによって得られます。そして、非同期で処理が可能な範囲は、アプリケーション毎に異なります。

しかしながら、このような疑問を持たれることは珍しいことではないらしく、コミュニティフォーラムでも、1つのスレッド（How to do bulk get in 2.x java sdk^{*3}）が立てられています。このスレッドには、有識者による回答として、非同期 API を内部的に用いたバルク処理メソッドの実装も示されています（Node.js ではなく、Java SDK による実装ですが）。

アプリケーションの要件を考慮せず、一律に同期を行う戦略は、必ずしも最善の選択であるとは言えず、だからこそ SDK は単純なラッパー API を提供していないとも言えますが、ここで挙げた論点に関する議論と、それに対する1つの解決策のサンプルとして興味深いと考えられたため、紹介します。興味のある方は、スレッドを参照してみてください。

^{*3} <https://forums.couchbase.com/t/how-to-do-bulk-get-in-2-x-java-sdk/4634/5>

17.3 データ一貫性 (Consistency)

はじめに、アプリケーションに必要なデータ一貫性 (Consistency) を実現するための論点を整理します。その後、関連する Couchbase Server の機能を解説します。

分散アーキテクチャーにおけるインデックス

Couchbase Server の特徴として、メモリーファーストアーキテクチャーがあります。このアーキテクチャーにより、永続化装置およびネットワークの性能が、データベースの性能のボトルネックになることが回避されます。ディスクへの書き込みは、キューを介して行われます。また、他ノードへの複製（レプリカ）のためのキューメカニズムとして、Data Change Protocol(DCP) が存在します。

DCP は、レプリカだけではなく、データの変更を他のサービスに反映するために広く利用されます。データ更新時のインデックスへの反映もその 1 つです。

インデックス更新から見た RDB との違い

データ更新時に、関連するインデックスの更新が行われるのは、Couchbase Server と RDB とで共通です。

RDB では、クライアントからのデータ更新処理リクエストに対して、成功のステータスが返されるのは、インデックス更新の終了後です。これは、データ一貫性の保証という要件面に加え、RDB がモノリシックなアーキテクチャーからなっているという技術面から見ても自然な挙動と言えます。

一方で、Couchbase Server は、分散アーキテクチャーという性格を持ち、データの更新処理を司る Data サービスと、インデックスを管理する Index サービスとは、コンポーネントとして独立しています。また、本番環

第17章 Node.js アプリケーション開発 17.3 データ一貫性 (Consistency)

境では、それらのサービスは別々のサーバーに配置されるのが一般的です。Couchbase Server では、インデックスの更新は、永続化装置への反映や、複製の作成と同様、キュー（非同期）のメカニズムで実現されています。

キーバリュー操作とクエリ

RDB にとって、SQL がクライアントにとって共通のインターフェースであるのと異なり、Couchbase Server では、N1QL は、クライアントにとって、選択することのできる複数のインターフェースのうちの 1 つです。

さらに言えば、Data サービスへのキーバリュー操作が、Couchbase Server における一次的なインターフェイスであるのに対して、N1QL クエリは、あくまで派生的なインターフェースであると言えます。キーによるアクセスでは、常にデータの同期が保証されます（たとえ、更新が永続化されておらず、ディスクキューに残っている状態だとしても、クライアントからのリクエストへの応答には、メモリ内の最新の情報が用いられます）。

N1QL クエリは、インデックスに依存するため、デフォルトでは、クエリの結果が最新のデータと同期されていることが保証されません（同期を保証する方法について後に触れます）。

インデックス更新が非同期であることの影響範囲

アプリケーションのバックエンドとしてのデータベースにとって、データ更新とインデックス更新の同期が必要な場合と、必要でない場合とを考えることができます。

RDB での開発に慣れている開発者にとって、データベースの中で、データとインデックスの同期が取れていない可能性を考えることは、違和感があるかもしれません。一方、経験が豊富な開発者ほど、インデックス定義によるデータ検索時の性能向上と、必然的に伴うデータ更新時の性能劣化との相克に悩まされた経験を持っているのではないかと思います。

第17章 Node.js アプリケーション開発 17.3 データ一貫性 (Consistency)

まず、インデックスは、あくまでデータの「検索」に使われることを確認しておきたいと思います。つまり、データの変更がインデックスに同期されていない場合の影響は、あくまで検索結果の違いにのみ関係してきます。ある特定のレコード（ドキュメント）のカラム（フィールド）の値について、データベース内で同期が取れていない（特定時点で、異なる複数の結果を参照しうる）という現象は起こりません。ここからみても、インデックスの更新が非同期であることの影響範囲は限定的であることが分かります。

また、ここでは非同期、つまり更新の遅延、を問題としていますが、クライアント（アプリケーション）から見た場合、この遅延は、究極的には、ネットワークその他のあらゆる要因による遅延と区別できないとも言えます。この後に触れる、ある1つのケースを除いては。

Read Your Own Writes (自分自身が書いたデータを読む)

複数のリクエストに対して矛盾した結果が返されたことを証明するためには、その矛盾の論拠として、正確な時刻情報を示す必要があります。これは、複数の異なるクライアントからのリクエストを想定した場合、遅延が余程大きいものでなければ、非常に困難と言えるでしょう（上に触れたように、クライアントとデータベースの間には様々な遅延要因が存在し得るため、原因がデータベース内の遅延によるものだと証明することはシステムの内部情報に当たらない限り不可能だと言えます）。言い換えれば、複数のクライアント間の出来事を考えた場合、多少の遅延は事実上の問題となりえないと言えるでしょう（ここでは、あくまで複数のクライアントが「継続的に」アプリケーションを利用し続けている状況について、語っています。これは、システムが利用されているほとんどの時間を占めている、と言えます。しかしながら、より頻繁ではないにしろ、「特定時点のデータ断面」の把握が必要になる要件は、当然あり得るでしょう。特定時点までのデータ同期を保証する方法については、後に触れます）。

Web アプリケーションでは、複数のクライアントからの複数のリクエス

第17章 Node.js アプリケーション開発 17.3 データ一貫性 (Consistency)

トと、同一のクライアントからの複数のリクエストが、さほど明確に区別されない場合があります。だからこそ、同一のクライアントからのシーケンシャルなリクエストとして「Read Your Own Writes (自分自身が書いたデータを読む)」ケースを、特別なものとして考慮する必要があります。自分で行った更新が、その後の検索結果に反映されないとしたら、これは明らかに問題です。

永続化や複製の場合と比べたインデックスの特殊性

データの永続化、複製、インデックス更新は、全て、データ更新とは非同期の処理であるという点で共通しています。一方、データ永続化および複製については、データ更新時に、同期（処理が完了するのを待ってクライアントに成功のステータスが返されること）を強制する永続性 (Durability^{*4}) オプションが存在しています。これに対して、インデックス更新については、このようなデータ更新時のオプションはありません（この違いは、永続化や複製が、データ更新と同じく、Data サービスの役割であることを考えると、自然と納得されます）。

一貫性を保証するためのオプション

データ更新時にインデックス更新の同期を強制するオプションが存在しない代わりに、N1QL クエリ実行時における、インデックスの一貫性 (Index Consistency^{*5}) に関する選択肢があります。

この選択肢は、以下のようなキーワードを用いて区別されます。

- **NOT_BOUNDED** クエリの一貫性を必要とせずに、クエリをすぐに実行します。インデックスの更新に遅延がある場合は、古い結果

^{*4} <https://docs.couchbase.com/server/current/learn/data/durability.html>

^{*5} <https://docs.couchbase.com/java-sdk/current/concept-docs/n1ql-query.html#index-consistency>

第17章 Node.js アプリケーション開発 17.3 データ一貫性 (Consistency)

が返されることがあります。

- **REQUEST_PLUS** インデックスが、クエリリクエストのタイムスタンプまで更新されるのを待って、クエリを実行します。
- **AT_PLUS** インデックスがクライアントの直近の更新のタイムスタンプまで更新されるのを待って、クエリを実行します。

はじめの 2 つについては、SDK によるクエリ実行時に、直接指定できるオプションがあり、オプションを指定しない場合のデフォルトは NOT_BOUNDED です。

上記のオプション表記は、概念を説明するためのものであり、SDK/プログラム言語によって実際の表記が変わることにご注意ください。以下は、Node.js SDK での REQUEST_PLUS オプションの利用例です。

```
const result = await cluster.query(
  "select * from `travel-sample`.inventory.airport",
  {
    scanConsistency: couchbase.QueryScanConsistency.RequestPlus,
  });

```

REQUEST_PLUS が、リクエストのタイムスタンプを基準にしているため、クライアント/アプリケーションの状態に依存していないのに対して、AT_PLUS は、本来の意味での「Read Your Own Writes」を実現します。AT_PLUS の実現には、データ更新時のトークンの取得が必要になります。取得したトークンをクエリ実行時に指定することによって、アプリケーションに必要な十分な一貫性を、最低限の負荷により、保証することができます。

■コラム: 本書におけるトークンの扱い、および参考情報
本書では、トークンの利用については扱いません。

第17章 Node.js アプリケーション開発 17.3 データ一貫性 (Consistency)

代わりに、サンプルコードが掲載されているドキュメント^{*6}を示します。

概略を述べると、トークンを利用するには、トークンの取得を有効化した上で、データ更新時のレスポンスから `MutationState` オブジェクトを取得し、クエリ実行時に、`MutationState` オブジェクトを引数に取るメソッド (`consistentWith`) を利用します。

■コラム：サンプルコード実装に関する注記

本番システムを考えると、リソース消費の面で悪手であることは間違いありませんが、`REQUEST_PLUS` を用いても、「Read Your Own Writes」のケースに対応することは可能です。本書のサンプルコードでは、こちらの方法を用いています。

付け加えると、リソース消費の最適化のためには、必要な時にのみ `A_T_PLUS`、ないし `REQUEST_PLUS` を用いるべきであり、必要な要件のない限り、`NOT_BOUNDED` を用いるべきです。その観点から、同じ検索処理であっても、データの更新後と、それ以外の通常の場合とでは、オプションを変更する、という実装を行うことが考えられます（本書のサンプルコードでは、区別なしに `REQUEST_PLUS` を用いています）。

^{*6} <https://docs.couchbase.com/nodejs-sdk/2.6/scan-consistency-examples.html>

第17章 Node.js アプリケーション開発7.4 サンプルアプリケーション紹介

17.4 サンプルアプリケーション紹介

ここで紹介するサンプルアプリケーションを、下記で公開しています。

https://github.com/YoshiyukiKono/Couchbase_Server_First_Step_Guide.git

CEAN スタック

ここで、用いている CEAN スタックとは、下記の技術要素からなります。

- C: Couchbase Server (NoSQL ドキュメント指向データベース)
- E: Express^{*7} (Web アプリケーションフレームワーク)
- A: Angular^{*8} (フロントエンドフレームワーク)
- N: Node.js^{*9} (サーバーサイド JavaScript 実行環境)

類似のものとして、MEAN スタックという言葉を聞いたことがある方もいるのではないかと思います。その場合の M は、MongoDB を指します。

アプリケーション概要

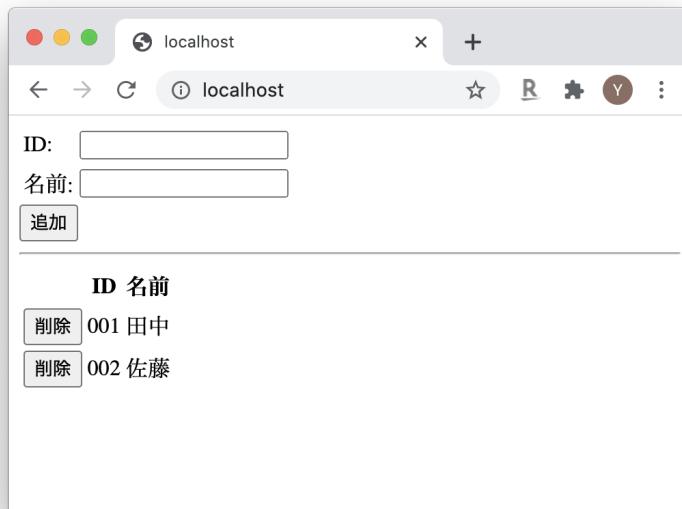
サンプルアプリケーションは、最小限の機能からなるシンプルなものとなっています。下記の画面を見ていただきさえすれば、特にそれ以上の説明は必要ないでしょう。

^{*7} <http://expressjs.com/>

^{*8} <https://angular.io/>

^{*9} <https://nodejs.org/>

第17章 Node.js アプリケーション開発7.4 サンプルアプリケーション紹介



アプリケーション実行事前準備

アプリケーションのデータ管理のために、以下の構成を前提としています。これらは事前に作成されている必要があります。

- パケット: `user_management`
- スコープ: `japan`
- コレクション: `users`

また、下記のインデックスを作成しておく必要があります。このサンプルアプリケーションでは、ユーザーのリストを表示する際に、検索条件を用い

第17章 Node.js アプリケーション開発7.4 サンプルアプリケーション紹介

ないクエリを利用しているためプライマリインデックスを作成しています。

```
CREATE PRIMARY INDEX primary_idx_users ON user_management.japan.users
```

アプリケーションが Couchbase Server にアクセスするために利用するアカウントを以下の内容で、新規作成します。

- ユーザー: user_management_app
- パスワード: C0uchb@se
- アクセス権: 上記バケット (user_management) に対する Application Access の権利を追加

アプリケーション利用方法

GitHub リポジトリをクローンし、作成されたディレクトリへ移動します。

```
$ git clone https://github.com/YoshiyukiKono/Couchbase_Server_First_S
$ cd Couchbase_Server_First_Step_Guide/
```

ディレクトリの中に含まれる、`package.json` には、下記の依存関係が定義されています。

```
"dependencies": {
  "couchbase": "^3.2.1",
  "express": "^4.17.1"
}
```

第17章 Node.js アプリケーション開発7.4 サンプルアプリケーション紹介

アプリケーションが依存するパッケージをプロジェクトにインストールします。

```
$ npm install
```

アプリケーションを実行します。下記のように、`server.js` を使います（または、`npm start` に置き換えることもできます）。

```
$ node server.js
Server up: http://localhost:80
```

`http://localhost:80` にアクセスします。

プログラム解説

Couchbase Serverへの接続

Couchbase Server は、アプリケーションと同じ環境で実行されていることが想定されています。

```
const cluster = new couchbase.Cluster('couchbase://localhost', {
  username: 'user_management_app',
  password: 'C0uchb@se',
})
```

ユーザーリスト表示

`routes.js` を見ると、ユーザーのリスト表示のために、下記のクエリが

第17章 Node.js アプリケーション開発7.4 サンプルアプリケーション紹介

使われているのが分かります。

```
const qs = 'SELECT id, name from users';
```

上記の(変数 `qs` に代入された)クエリを実行している箇所は以下の通りです。

```
const selectUsers = async (key) => {
  const result = await scope.query(qs, {
    scanConsistency: couchbase.QueryScanConsistency.RequestPlus,
  });
  return result.rows;
}
```

`script.js` の中で、上記クエリのリクエストの戻り値 (`response.data`) は、そのまま変数 (`$scope.users`) に代入されます。

```
var getData = function() {
  return $http({
    method: 'GET',
    url: '/users'
  }).then(function successCallback(response) {
    $scope.users = response.data;
    console.log('Success: ' + response);
  }, function errorCallback(response) {
    console.log('Error: ' + response);
  });
};
```

`index.html` の中で、代入された変数 (`users`) は、下記のように画面表示

第17章 Node.js アプリケーション開発7.4 サンプルアプリケーション紹介

に利用されます。

```
<tr ng-repeat="user in users">
    <td><input type="button" value="削除" data-ng-click="del_user(user)"></td>
    <td>{{user.id}}</td>
    <td>{{user.name}}</td>
</tr>
```

新規ユーザー追加

routes.js の中の、データ（ドメインオブジェクト）の表現と、そのデータをデータベースへの保存する部分は、以下の通りです。

```
const user = {
  id: req.body.id,
  name: req.body.name,
};
upsertDocument(user);
```

以下の箇所を見ると、JavaScript のディクショナリが、そのままドキュメントとして格納されているのが分かります。

なお、既存の ID でユーザーを追加した場合には、新しい情報で上書きされる仕様としています。

```
const addUser = async (doc) => {
  try {
    const key = `${doc.id}`;
    const result = await collection.upsert(key, doc);
  } catch (error) {
```

第 17 章 Node.js アプリケーション開発5 Node.js SDK を開発に利用する

```
        console.error(error);
    }
};
```

ユーザー削除

ドキュメントの削除は、キー指定により行われます。

```
const removeUser = async (id) => {
  try {
    const key = `${id}`;
    const result = await collection.remove(key);
  } catch (error) {
    console.error(error);
  }
};
```

17.5 Node.js SDK を開発に利用する

新しく作成する Node.js アプリケーションで、Couchbase Server の Node.js SDK を利用するための手順を記します。

動作確認環境

- Mac OS Catalina Version 10.15.6
- Node.js v15.4.0
- npm 7.0.15

第 17 章 Node.js アプリケーション開発5 Node.js SDK を開発に利用する

プロジェクト準備手順

プロジェクト用のディレクトリを作成し、その中で作業を開始します。
はじめにプロジェクトを初期化します。

```
$ npm init -y
```

package.json ファイルが作成されます。

次に、Node.js SDK をインストールし、package.json に追記します。

```
$ npm install couchbase --save
```

package.json に以下のような依存関係が追記されます。

```
"dependencies": {  
    "couchbase": "^3.2.1"  
}
```

第17章 Node.js アプリケーション開発ODM フレームワーク Ottoman.js

17.6 ODM フレームワーク Ottoman.js

Ottoman.js とは

Ottoman.js^{*10}は、Couchbase Server と Node.js のための ODM(オブジェクトデータモデル)です。

Node.js アプリケーション開発に Couchbase Server を利用する際、Ottoman の利用は必須ではありませんが、Ottoman は、開発者に様々な恩恵をもたらすことを目的として開発されています。^{*11}

Ottoman は、MongoDB における Mongoose ODM^{*12}に相当するものであると言えます。

機能概要

Couchbase Server では、RDB のようにデータに対してスキーマが強制されることはありません。しかしながら、アプリケーション開発において、スキーマは依然として重要な役割を持ちます。

Ottoman は、アプリケーションサイドにおけるスキーマによるデータバリデーション(検証)のフレームワークを提供します。

また、Ottoman は、Couchbase Server SDK に対する抽象化レイヤーを提供します。Ottoman ライブラリが使われる時、Ottoman を介して、透過的に Couchbase Server とのデータのやり取りが行われます。

*10 <https://ottomanjs.com/>

*11 Ottoman(オットマン)の元々の意味は、椅子やソファーの前において、足を乗せるために使う小型のソファーのことです。フットスツールとも呼ばれます。名前は、発祥地であるオスマン帝国に由来します。Ottoman.js の命名については、カウチ(ソファー)と組み合わせて利用することから来ているのでしょう。

*12 <https://mongoosejs.com/>

■コラム: Ottoman 関連情報

Ottomanについては、当初本書では扱うことを考えていませんでしたが、MongoDBのMongoose ODM(の存在)を知っている人が持つかもしれない疑問/興味に応えるために、紹介を目的として、ごく基本的な情報のみ記しています。

ここでは、Ottomanに関心を持たれた方向けに、追加の情報を提供します。

Ottomanは、2021年9月に、v2.0がリリースされています。

公式サイトには、クイックスタート(Quick Start with Ottoman v2)が公開されており、利用を始める際の参考にすることができます。

また、v2.0のリリースにあわせて、ブログ記事(「Introducing Ottoman v2.0: An ODM for Node.js & Couchbase」^{*13})が発表されています。この記事は、v2.0の新規機能のみではなく、Ottomanそのものへのイントロダクションとなっているため、Ottomanを使って、何ができるのかを理解するために有用だと思われます。

また、サンプルアプリケーション^{*14}も公開されており、v2.0リリースに合わせてアップデートされています。

17.7 開発の実践に向けて

本書によって、Couchbase Serverを使った開発について、基本的なイメージを掴むことができたと感じていただけたのなら幸いです。

一方、当然のことながら、アプリケーション開発は、一冊の書籍で語り尽

^{*16} <https://blog.couchbase.com/ottoman-2-0-odm-node-js-couchbase/>

^{*17} <https://github.com/couchbaselabs/try-ottoman>

くせてしまうほど単純なものではなく、困難を伴うが故に奥深く、興味深いものです。

ここでは、さらに学習を進めるにあたって有益と思われる情報を紹介し、締め括りとしたいと思います。

コミュニティ

Couchbase Dev Community サイト^{*15}では、Couchbase 開発者コミュニティへ向けて、チュートリアルやベストプラクティス等、様々な情報提供が行なわれています。また、フォーラム^{*16}で、Couchbase Server コミュニティエディションに関する疑問について、過去に行なわれた質疑応答を検索したり、質問をすることができます。

Couchbase Labs とサンプルアプリケーション

Couchbase Labs GitHub^{*17}では、Couchbase に関する様々なプロジェクトが公開されています。

Ottoman のようなフレームワークの他、サンプルアプリケーションも公開されています。例えば、travel-sample サンプルバケットを使ったアプリケーションが公開されており、フロントエンドは、Node.js で実装されています。^{*18} そのフロントエンドと組み合わせて利用することができるバックエンドの REST API アプリケーションは、各種プログラミング言語での実装が公開されており、Node.js SDK による実装も存在します。^{*19}

^{*15} <https://developer.couchbase.com/>

^{*16} <https://forums.couchbase.com/>

^{*17} <https://github.com/couchbaselabs/>

^{*18} <https://github.com/couchbaselabs/try-cb-frontend-v2>

^{*19} <https://github.com/couchbaselabs/try-cb-nodejs>

無償オンライントレーニング

Couchbase, Inc. が運営する Couchbase Academy^{*20}では、各種のトレーニングコースや認証資格が提供されています。自分のペースで受講することができる、Node.js SDK についての無償のオンライントレーニングとして、「CB130n: Couchbase Associate Node.js Developer Certification Course^{*21}」が利用できます（対応する認証資格試験の受講は有償になります）。

ドキュメント、API Docs、ソースコード

ドキュメント^{*22}と API Docs^{*23}は、常に開発者の良き友です。さらにお望みとあらば、ソースコード^{*24}を参照することもできます。

^{*20} <https://www.couchbase.com/academy>

^{*21} <https://learn.couchbase.com/store/1246860-cb130n-couchbase-associate-node-js-developer-certification-course>

^{*22} <https://docs.couchbase.com/nodejs-sdk/current/hello-world/overview.html>

^{*23} <https://docs.couchbase.com/sdk-api/couchbase-node-client/modules.html>

^{*24} <https://github.com/couchbase/couchnode>

付録 A

NoSQL 性能評価: MongoDB、Cassandra と の比較

以下の NoSQL データベースに対して実施されたベンチマークの内容を紹介します。

- MongoDB v3.6
- DataStax Enterprise v6 (Cassandra)
- Couchbase Server v5.5

ここで紹介するベンチマークは、2018 年に Altoros^{*1}社によって実施されたもので、下記のタイトルで公開されています。

NoSQL Performance Benchmark 2018: Couchbase Server v5.5, DataS-

^{*1} <https://www.altoros.com/>

付録 A NoSQL 性能評価: MongoDB、Cassandra & の比較ベンチマーク概要

tax Enterprise v6 (Cassandra), and MongoDB v3.*²

ここでは、このベンチマークについて、引用を行いながら、全体の要約を試みます。要約に際して、偏った解釈を交えないよう心掛けていますが、正確な内容や詳細の確認については、原本に当たっていただきますようお願いします。

また、Altoros 社によって Couchbase Server の検証に用いられた方法が不適切と考えられる箇所で、指摘を行います。

A.1 ベンチマーク概要

検証パターン

検証のパターンとして、2種類の YCSB(Yahoo! Cloud Serving Benchmark)^{*3}標準ワークロードと、2種類の独自のクエリが用いられています。

- ワークロード A: 更新処理
- ワークロード E: レンジスキヤン
- クエリ 1: ページネーション(OFFSET と LIMIT によるフィルター)
- クエリ 2: ジョイン(テーブル結合)

クエリ 1 とクエリ 2 は、それぞれ下記のドメインとシナリオを表現しています。

- 財務: 地域を指定した財務データの一覧表示(ブラウザー表示のためのページネーションを考慮)
- e コマース: 期間と地域を指定した売り上げ集計レポート

*² <https://www.altoros.com/research-papers/nosql-performance-benchmark-2018-couchbase-server-v5-5-datastax-enterprise-v6-cassandra-and-mongodb-v3-6/>

*³ <https://en.wikipedia.org/wiki/YCSB>

付録 A NoSQL 性能評価: MongoDB、Cassandra 及びの比較マーク概要

クラスター構成

サイズの異なる以下の 3 通りのクラスターで、それぞれ異なるサイズのデータを用いて、検証が実施されています。

- 4 ノード
- 10 ノード
- 20 ノード

サーバーハードウェアスペック

検証環境として、AWS(Amazon Web Services) が利用され、クラスターを構成するサーバーとして、下記スペックの EC2 インスタンスが用いられています。

インスタンスタイプ	i3.2xlarge (Storage Optimized)
vCPUs	8
メモリー	61 GB
インスタンスマニフェストレージ	1,900 GB (SSD)
ネットワークパフォーマンス	最大 10 GB
オペレーティングシステム	Ubuntu 16.04 LTS

共通条件

- データサイズは、メモリーサイズと適合している状態
- 各データセットに対して 1 つのレプリカ (複製)

データベース固有の構成

Couchbase Server は、本番環境ではノード毎に異なるサービスを割り当てることが一般的ですが、ここでは以下の簡略化された構成が用いられています。

付録 A NoSQL 性能評価: MongoDB、Cassandra の検証較ターン別詳細

います。

- クラスターのサイズを問わず、全てのノードで、Data サービス、Index サービス、Query サービスを有効化。
- Data サービスには、データベースに使用可能な RAM の 60% を割り当て。
- Index サービスには、データベースに使用可能な RAM の 40% を割り当て。インデックスはすべてのノードに複製。

MongoDB は、Router プロセス、Config サーバー、およびデータシャードからなる階層型クラスター構成を持っていますが、ここでは以下の構成が使用されています。

- Config サーバーは、3 つのメンバーからなるレプリカセットとして構成（クラスターのノード数としてはカウントしない、別のサーバーを利用）。
- 各シャードは、3 つのメンバー（Primary、Secondary、Arbeiter）からなるレプリカセットとして構成。
- mongos（Router プロセス）は、クライアント環境にデプロイ。

Cassandra のクラスター構成については、ノード毎の差異はなく、全てのノードに共通の設定が適用されています。

A.2 検証パターン別詳細

以下、各検証パターン毎に、下記の情報を整理します。

- データアクセス特性
- 利用データ件数
- 評価用クエリ
- 測定結果

付録 A NoSQL 性能評価: MongoDB、Cassandra の検証較時間別詳細

ワークロード A: 更新処理

データアクセス特性

- 読み取り: 50%
- 更新: 50%

表 A.1: ワークロード A 利用データ件数

クラスター構成	テストデータ
4 ノード	5 千万件 (1KB/1 件)
10 ノード	1 億件
20 ノード	2 億件

Table 4.1 Evaluated queries for Workload A

Couchbase N1QL	MongoDB Query	Cassandra CQL
bucket.get(docId, RawJsonDocument.class)	db.ycsb.find({_id: \$1})	SELECT * FROM table WHERE id = \$1 LIMIT 1

図 A.1: ワークロード A 評価用クエリ

付録 A NoSQL 性能評価: MongoDB、Cassandra の検証較時間別詳細

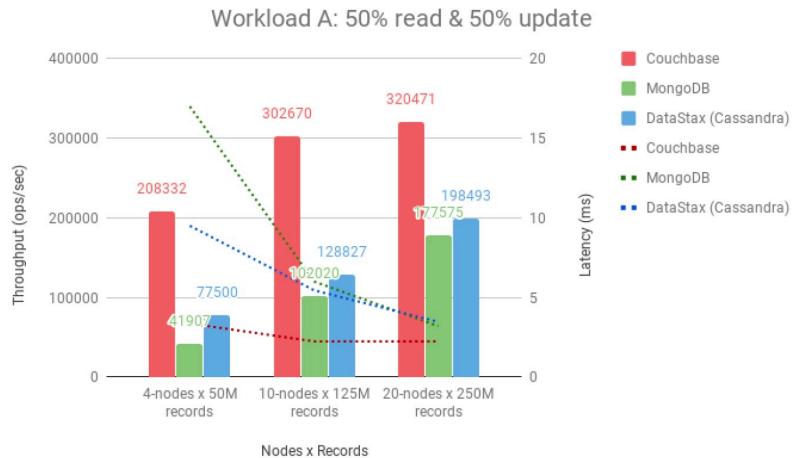


Figure 4.1.3 Performance results under Workload A on 4-, 10-, and 20-node clusters

図 A.2: ワークロード A 測定結果

ワークロード E: レンジスキヤン

データアクセス特性

- 読み取り: 95%
- 更新: 5%

表 A.2: ワークロード E 利用データ件数

クラスター構成	テストデータ
4 ノード	5 千万件 (1KB/1 件)
10 ノード	1 億件
20 ノード	2 億 5 千万件

付録 A NoSQL 性能評価: MongoDB、Cassandra の検証較ターン別詳細

Table 4.2 Evaluated queries

Couchbase N1QL	MongoDB Query	Cassandra CQL
<pre>SELECT RAW meta().id FROM `ycsb` WHERE meta().id >= \$1 ORDER BY meta().id LIMIT \$2</pre>	<pre>db.ycsb.find({ _id: { \$gte: \$1 }, _id: 1 }).sort({ _id: 1 }).limit(\$2)</pre>	<pre>SELECT id FROM table WHERE token(id) >= token(\$1) LIMIT \$2</pre>

図 A.3: ワークロード E 評価用クエリ

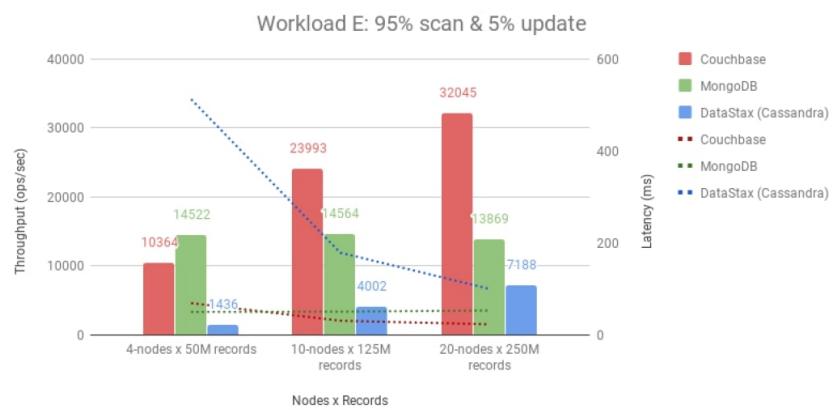


Figure 4.2.3 Performance results under Workload E on 4-, 10-, and 20-node clusters

図 A.4: ワークロード E 測定結果

冒頭に記した、「Altoros 社によるベンチマーク方法が不適切と考えられる箇所」は、ワークロード E に関係しています。原本中の解説では、以下のインデックスを利用したとされています。

付録 A NoSQL 性能評価: MongoDB、Cassandra の検証較ターン別詳細

```
CREATE PRIMARY INDEX 'ycsb_primary' ON 'ycsb'  
USING GSI WITH {"nodes": [...]};
```

ここには、Couchbase Server のプライマリインデックスに対する誤解が見えます。

原本には、「スキャン操作は Couchbase の主キーに対して実行されるため (Because the scan operation is performed over the primary key in Couchbase)」このプライマリインデックスが作成された、とされています。しかしながら実際には、Couchbase Server において、主キーに相当するものは、ドキュメントキーであり、Couchbase Server のプライマリインデックスは、主キーとは直接関係がないものです。Couchbase Server のプライマリインデックスは、そのキースペース（この検証では、ycsb バケット）に対する、ドキュメントの特定のフィールドでの検索に限定されないクエリを可能にする（そのキースペースに対する全件スキャンを行うことを可能にする）ためのものであって、ドキュメントキーを使った検索を最適化するものではありません。

検証に用いられたクエリの実行を最適化するためには、以下のように、実際にクエリの WHERE 句で用いられている META().id を指定した（セカンダリー）インデックスを作成する必要があります。

```
CREATE INDEX ycsb_secondary on ycsb(META().id);
```

性能測定結果においても、次の「ページネーション」の検証と比べた時、最も効率が良いはずである ID を用いた検索よりも、通常のデータの値を用いた検索の方が、遙かに高い性能が出ていることが分かります。これは、明らかに矛盾しており、それを裏付けるように、他のデータベースでは、前者

付録 A NoSQL 性能評価: MongoDB、Cassandra の検証較ターン別詳細

が後者に勝るという納得できる結果となっています。この違いは、後者の検証では、(原本に記載されている通り) 下記のような適切な(セカンダリー)インデックスが作成されていることから来ています。

```
CREATE INDEX 'ycsb_address_country' ON 'ycsb' (address.country)
USING GSI WITH {"nodes": [...]}
```

クエリ 1: ページネーション (OFFSET と LIMIT によるフィルター)

データアクセス特性

- 読み取り: 100%

表 A.3: ページネーションワークロード利用データ件数

クラスター構成	テストデータ
4 ノード	5 百万件の顧客データ (4KB/1 件)
10 ノード	2 千 5 百万件の顧客データ
20 ノード	5 千万件の顧客データ

付録 A NoSQL 性能評価: MongoDB、Cassandra の検証較ターン別詳細

Table 4.3 Evaluated queries

Couchbase N1QL	MongoDB Query	Cassandra CQL
<pre>SELECT RAW meta().id FROM `ycsb` WHERE address.country='\$1' OFFSET \$2 LIMIT \$3</pre>	<pre>db.customer.find({ address.country: \$1 }, { _id: 1 }) .skip(\$2) .limit(\$3)</pre>	<pre>SELECT id FROM table WHERE address_country = \$1 LIMIT \$2</pre>

図 A.5: ページネーションワークロード評価用クエリ

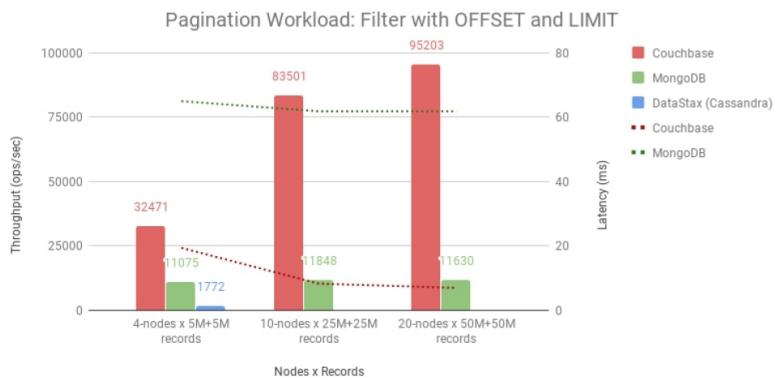


Figure 4.3.3 Performance results under the Pagination Workload on 4-, 10-, and 20-node clusters

図 A.6: ページネーションワークロード測定結果

クエリ 2: JOIN (テーブル結合)

データアクセス特性

- 読み取り: 100%

付録 A NoSQL 性能評価: MongoDB、Cassandra の検証較ターン別詳細

表 A.4: JOIN ワークロード利用データ件数

クラスター構成	テストデータ
4 ノード	5 百万件の顧客データ (4KB/1 件)、5 百万件の受注データ (4.5KB/1 件)
10 ノード	2 千 5 百万件の顧客データ、2 千 5 百万件の受注データ
20 ノード	5 千万件の顧客データ、5 千万件の受注データ

Table 4.4 Evaluated queries

Couchbase N1QL	MongoDB Query	Cassandra CQL
<pre> SELECT o2.month, c2.address.zip, SUM(o2.sale_price) FROM `ycsb` c2 INNER JOIN `ycsb` o2 ON (META(o2).id IN c2.order_list) WHERE c2.address.zip = \$1 AND o2.month = \$2 GROUP BY o2.month, c2.address.zip ORDER BY SUM(o2.sale_price) </pre>	<pre> \$rl = db.customer.find({ address.zip: \$1 }, { address.zip: 1, order_list: 1 }) \$rl = db.order.aggregate([{ \$match: { \$and: [{ _id: { \$in: \$rl.order_list } }, { month: \$2 }] }, { \$group: { _id: null, sum: { \$sum: "\$sale_price" } } }]) </pre>	<pre> SELECT month, zip, SUM(sale_price) FROM customer_orders_join WHERE zip = \$1 AND month = \$2 </pre>

図 A.7: JOIN ワークロード評価用クエリ

付録 A NoSQL 性能評価: MongoDB、Cassandra の検証較ターン別詳細

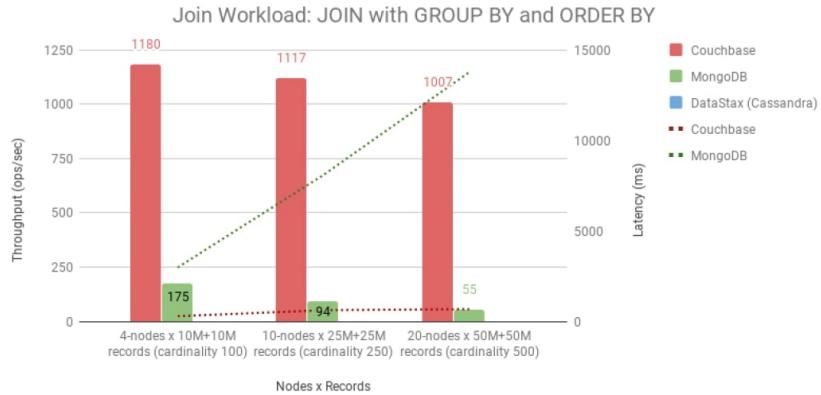


Figure 4.4.3.1 Performance results under Join Workload on 4-, 10-, and 20-node clusters

図 A.8: JOIN ワークロード測定結果 (オペレーションベース)

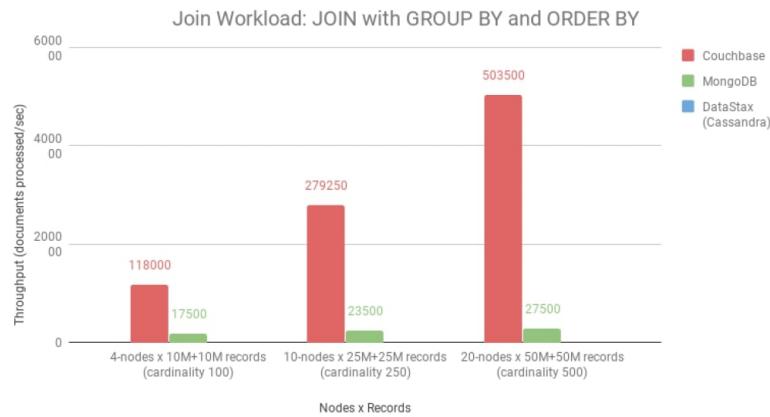


Figure 4.4.3.2 Number of documents processed under Join Workload on 4-, 10-, and 20-node clusters

図 A.9: JOIN ワークロード測定結果 (ドキュメントベース)

A.3 結論

以下、結論部から引用します。

Hardly any NoSQL database can perfectly fit all the requirements of any given use case. Every solution has its advantages and disadvantages that become more or less important depending on specific criteria to meet.

First of all, it should be noted that all the workloads were executed with the assumption that a data set fits the available memory. With that in mind, all the reads from Data Service and Index Service for Couchbase were from RAM and thus performed on in-memory speeds.

With the same amount of available RAM, DataStax (Cassandra) did not allow to store everything in cache. Therefore, the majority of the reads were made from disk. Couchbase demonstrated good performance across all the evaluated workloads and appears to be a good choice, providing out-of-the-box functionality sufficient to handle the deployed workloads and requiring no in-depth knowledge of the database's architecture. Furthermore, the query engine of Couchbase supports aggregation, filtering, and JOIN operations on large data sets without the need to model data for each specific query. As clusters and data sets grow in size, Couchbase ensures a satisfactory level of scalability across these operations. MongoDB produced comparatively decent results on relatively small clusters. MongoDB is scalable enough to handle increasing amounts of data and cluster extension. Under this benchmark, the one issue we observed was that MongoDB did not support

付録 A NoSQL 性能評価: MongoDB、Cassandra との比較 A.3 結論

JOIN operations on sharded collections out of the box. This way, dedicated data modeling provided a way out—however, with a negative impact on performance.

Cassandra provided rather good performance for intensive parallel writes and reads by a partition key and, as expected, failed on non-clustering key-read operations. In general, we proved that Cassandra is able to show great performance for write-intensive operations and reads by a partition key. Still, Cassandra is operations-agnostic and behaves well only if multiple conditions are satisfied. For instance, reads are processed by a known primary key only, data is evenly distributed across multiple cluster nodes, and, finally, there is no need for joins or aggregates.

以下、上記引用からの翻訳を示します。

NoSQL データベースが、任意のユースケースのすべての要件に完全に適合することは稀です。すべてのソリューションには長所と短所があり、満たすべき基準に応じて重要性が異なります。

まず、すべてのワークロードは、データセットが使用可能なメモリに適合することを前提として実行されました。そのため、Couchbase の Data サービスと Index サービスからの読み取りはすべて RAM から行われ、インメモリの速度で実行されました。

使用可能な RAM は同じ量ながら、DataStax (Cassandra) では、すべてをキャッシュに保存することができませんでした。したがって、読み取りの大部分はディスクから行われました。Couchbase は、評価されたすべてのワークロードで優れたパフォーマンスを示し、適切な選択であるように伺われました。デプロイされたワークロードを処理するのに十分な機能が提供されており、それらを特別な設定を行わずとも利用することができたため、データベースのアーキテクチャーに関する深い知識は必要ありませんでした。さらに、Couchbase の

付録 A NoSQL 性能評価: MongoDB、Cassandra との比較 A.3 結論

クエリエンジンは、特定のクエリに合わせてデータをモデル化する必要なく、大規模なデータセットに対する集計、フィルタリング、および JOIN 操作をサポートしています。Couchbase は、大規模のクラスターと大量のデータセットを用いた検証において、オペレーション全般について十分なレベルのスケーラビリティを実証しました。

MongoDB は、小さなクラスターでの検証では、比較的適切な結果を生成しました。MongoDB は、増加するデータ量をクラスターの拡張により処理することにおいて十分スケーラブルだといえます。このベンチマークの下で、私たちが観察した 1 つの問題は、MongoDB がシャードコレクションでの JOIN 操作を、素のままの状態では、サポートしていないことでした。そのため、専用のデータモデリングを行わねばならず、このことは性能に悪影響を与えました。

Cassandra は、パーティションキーによる集中的な並列書き込みと読み取りに対してかなり優れたパフォーマンスを提供し、予期されたことでしたが、非クラスタリングキー読み取り (non-clustering key-read) 操作で失敗しました。一般に、Cassandra は、書き込み集中型 (write-intensive) 操作とパーティションキーによる読み取りに対して優れたパフォーマンスを発揮できることを証明しました。一方、operations-agnostic な性格を持つ Cassandra は複数の条件が満たされた場合にのみ正常に動作します。たとえば、読み取りは既知の主キーによってのみ処理され、データは複数のクラスターノードに均等に分散され、そして最後に、結合や集計は必要とされていません。

付録 B

データマイグレーションツール

Couchbase Server と RDB との間でデータマイグレーションを行うツールが、サードパーティーから、あるいはオープンソースプロジェクトとして、提供されています。

このような外部ツールを用いなくとも、RDB から CSV 形式でデータを出力し、Couchbase Server が提供しているツール (`cbimport`) を使って、データマイグレーションプロセスを構築することができます。

一方、このような既存のツールの存在を押さえておくことも決して無意味ではないと考え、ここに紹介します。

B.1 GlueSync

GlueSync^{*1}は、MOLO17^{*2}社によって提供される「The only Couchbase-native data streaming connector for relational databases」です。Microsoft SQL Server または Oracle と Couchbase Server との間でリ

^{*1} <https://gluesync.com/>

^{*2} <https://molo17.com/>

アルタイムデータ同期を実現します。

以下、サイトから引用します。

GlueSync allows you to replicate data to and from RDBMS (Microsoft SQL Server and Oracle) in real time with Couchbase. Whether the goal is to Data Offload or increase the potential of your Business Apps, GlueSync is the only zero impact plug&play tool on your infrastructure.

B.2 Couchgres

Couchgres^{*3}は、オープンソースプロジェクトであり、その名の通り、Couchbase と PostgreSQL とのデータマイグレーションを実現します。

以下、GitHub リポジトリの README から引用します。

These are some small tools for easier migration between PostgreSQL and Couchbase 7.0, and visualising a Couchbase bucket with scopes and collections in the form of a pie chart.

B.3 SqlServerToCouchbase

SqlServerToCouchbase^{*4}は、SQL Server から Couchbase へのデータ移動のためのライブラリを提供するオープンソースプロジェクトです。

以下、GitHub リポジトリの README から引用します。

A library to help guide your SQL Server to Couchbase migration/sync efforts. This library represents a best effort at mapping

^{*3} <https://github.com/metonymic-smokey/couchgres>

^{*4} <https://github.com/mgroves/SqlServerToCouchbase>

付録 B データマイグレーションツール **B.3 SqlServerToCouchbase**

SQL Server concepts to Couchbase Server concepts automatically. It may or may not fulfill every one of your requirements. In the worst case, it can at least be an educational tool. Think of it like Google Translate.

このツールを紹介しつつ、ツールの紹介に終わらない、RDB から Couchbase Server へのマイグレーションに関する論点を整理したブログ記事「SQL to NoSQL: Automated Migration」^{*5}が発表されています。

^{*5} <https://blog.couchbase.com/sql-to-nosql-automated-migration/>

NoSQL ドキュメント指向データベース Couchbase Server ファーストステップガイド

2021 年 10 月 1 日 初版第 1 刷 発行

著 者 河野泰幸
