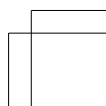
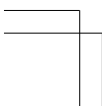
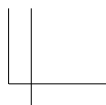
The page features decorative corner elements consisting of thin black lines forming L-shapes at the top-left, top-right, bottom-left, and bottom-right corners, as well as a central cross-like shape formed by a horizontal and a vertical line.

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

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、その含意は、必ずしもこの章が存在しなくても書籍全体の構成は成立する、という位置づけによるものです（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）。この章だけが、二人の登場人物の対話の形式で書かれています。この章では、特に重要な情報を、効率的に、そして願わ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「はじめに」が続きます。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ために、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多くの紙幅を費やしています。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、この章は、それぞれの節が基本的に独立している、という構造を持っています。

第4章以降については特に注釈は不要と考えます。章や節のタイトルを参照し、必要に応じ適当な箇所をご利用いただけるものと思います。

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。したがって、本書を用いた開発、制作、運用は、必ずご自身の責任と判断によって行ってください。本書の情報による開発、制作、運用の結果について、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7月25日まで開催された技術系同人誌即売会「技術書展 11」で頒布されたものを底本としています。

目次

まえがき	iii
本書の構成	iii
表記関係について	iv
免責事項	iv
底本について	iv
 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	 9
2.1 Couchbase について	9
Couchbase Server について	10
エディションとオープンソースプロジェクト	10
2.2 関連データベース技術	11
ドキュメントデータベース	12
キーバリューストア	12
インメモリデータベース	12
リレーショナルデータベース	13
クエリ	13
トランザクション	13
 第 3 章 Couchbase Server とは何か？	 15

目次

3.1	Couchbase Server の歴史	15
	Couchbase, Inc. の設立	15
	Couchbase Server の発展	16
3.2	NoSQL と現在の技術動向	19
	DX と SoE	19
	NoSQL の登場まで	21
	SoE と NoSQL	21
	SoE のためのデータベースとしての Couchbase Server . . .	23
3.3	メモリーファーストアーキテクチャー	24
	インメモリ DB と永続化	24
	データ永続化パターン	24
	データ復旧アプローチ	25
	インメモリ DB	26
	永続化機構を備えたインメモリ DB としての Couchbase Server	26
3.4	JSON データベース	28
	JSON、その歴史と関連技術	28
	JSON とモダンな開発・サービス形態との関係	30
3.5	トランザクション今昔	31
	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 .	31
	マルチドキュメントトランザクション	31
	NewSQL	33
	楽観的ロックの何が問題か	35
3.6	第一正規化制約を超えて	36
	NoSQL のカテゴリーについて	36
	「準（半）構造化データ」という呼称について	36
	RDB のスーパーセットとしての Couchbase Server	37

第 4 章 Couchbase Server を使ってみる 39

目次

4.1	環境構築手順	39
	インストーラー利用	39
	Docker 利用	40
	クラスター初回構築	44
	クラスター構成変更	50
4.2	管理コンソールの使い方	53
	バケット作成	53
	ユーザー作成	56
	データ登録	56
	クエリ実行	57
	インデックス作成	58
第 5 章	データ	59
5.1	データ管理	59
	ドキュメント	59
	バケット	60
	メモリからのデータ排出 (Ejection)	62
	監視と警告	65
	ドキュメント有効期限	67
5.2	スコープとコレクション	68
	概要	68
	存在意義	68
	機能	69
	種類と命名規則	70
第 6 章	クエリとインデックス	73
6.1	インデックス	73
	用語法に関する注釈	73
	来歴	74

目次

	RDB のインデックスとの違い	75
	ストレージモデル	76
	プライマリインデックス	77
	セカンダリインデックス	79
	複合 (コンボジット) インデックス	81
	ドキュメントキーのインデックス	81
	インデックスのメタデータ	82
6.2	インデックス設計	84
	カバリングインデックス	84
	パーシャル (部分) インデックス	84
	関数 (Functional) インデックス	85
	重複 (Duplicate) インデックス	86
	配列インデックス	87
	アダプティブ (Adaptive) インデックス	92
	性能最適化における考慮点	93
	複数ドキュメント取得時の最適化	94
6.3	N1QL クエリ	95
	基本	96
	構造	96
	ドキュメントキーの利用	97
	フィールドから値のみを取り出す: RAW ELEMENT VALUE	99
	配列からデータを取り出す	101
	配列を検索条件に用いる	103
	ドキュメントキーによる結合: NEST	104
	サブドキュメントのフラット化: UNNEST	106
	データ型: NULL, MISSING	108
	比較演算: IS [NOT] NULL MISSING	110
	比較演算: IS [NOT] VALUED	111
	SQL と比べた場合の制約	112

目次

6.4	結合	113
	ANSI JOIN	113
	Lookup JOIN	114
	Index JOIN	115
6.5	レンジスキャン	116
	分散データベース一般のレンジスキャン	116
	Couchbase Server の特殊性	116
	N1QL によるレンジスキャン	117
	Data サービス API によるレンジスキャン	119
6.6	ユーザー定義関数	120
第 7 章	アーキテクチャー	123
7.1	分散システムアーキテクチャー	123
	分散アーキテクチャーの特徴	123
	クラスターの構成変更	124
	ワークロード分散	125
	レプリケーション	126
	データの分散管理 (シャード)	127
	スマートクライアント	128
	データ転送	129
7.2	XDCC(Xross Data Center Replication)	130
	XDCC は、何故必要か?	130
	XDCC は、どのように動作するか?	131
	機能ハイライト	132
	運用上の注意	133
第 8 章	クラスター	135
8.1	フェイルオーバー	135
	分散アーキテクチャーのフェイルオーバー	135

目次

	Couchbase Server フェイルオーバー機能	140
	サービス固有の自動フェイルオーバーポリシー	141
	グループフェイルオーバー	142
	ディスク障害に対するフェイルオーバー	143
	フェイルオーバー実行方法	143
8.2	コミュニケーション概要	144
	クライアントとクラスター間	144
	ノード間	145
	クラスター間	145
	コネクター	145
8.3	クライアント・クラスター間コミュニケーション	145
	クライアント接続における 3 つのフェーズ	145
	他の分散アーキテクチャーとの違い	147
第 9 章	運用	149
9.1	監視	149
	Couchbase Server 監視機能	149
	外部監視システム連携	150
9.2	バックアップ: cbbakupmgr	152
	バックアップリポジトリの作成	153
	バックアップ構成の確認	154
	バックアップの作成	155
	増分バックアップ	157
	バックアップの復元	159
	バックアップのマージ	161
	バックアップリポジトリの削除	162
第 10 章	セキュリティ	165
10.1	Couchbase Server セキュリティ概観	165

目次

	認証 (Authentication)	165
	認可 (Authorization)	165
	監査 (Auditing)	166
	暗号化 (Encryption)	166
	その他	167
10.2	認証 Authentication	168
	ビルトイン管理者アカウント	169
	外部アクセシビリティ管理	169
	Web コンソールでの認証	170
	アプリケーションの認証	170
	証明書ベースの認証	171
	SASL	172
	パスワードポリシー設定	172
10.3	認可 Authorization	173
	ロールの基本的なユースケース	173
	Couchbase Server のロール	174
第 11 章	コマンドライン操作	177
11.1	コマンド	177
	couchbase-cli	177
11.2	ストレスツール	180
	インストール	180
	共通オプション	181
	pillowfight	182
	n1qlback	185
第 12 章	Couchbase Server の構成要素	189
12.1	Data サービス	189
	構成	190

目次

12.2	Query サービス	192
	構成	192
	クエリの実行	193
12.3	Index サービス	193
	構成	194
	他サービスとの連携	195
	ストレージ	196
12.4	クラスターマネージャー	197
	ns-server	197
	babysitter	198
	マスターサービスの役割	198
	クラスターメンバー管理	199
	バケット管理	200
	管理、統計、およびロギング	201
第 13 章	構成管理	203
13.1	コンパクション	203
	メカニズム	203
	運用における留意点	204
	オート・コンパクション	204
	マニュアル実行	207
13.2	IP アドレスのバインド	207
	クラスターマップとの関係	207
	代替 (Alternate) アドレス	208
	代替アドレスの割り当て方法	211
第 14 章	外部システム連携	213
14.1	コネクタ	213
第 15 章	Node.js アプリケーション開発	215

目次

15.1	なぜ、JavaScript と NoSQL の組み合わせなのか？	215
15.2	NoSQL/Couchbase を選択する理由	216
	JSON データの利点	216
	クエリ言語の利点	216
15.3	キーバリューストア操作	217
	CRUD 操作	217
	サブドキュメント操作	218
15.4	データ一貫性 (Consistency)	221
	分散アーキテクチャーにおけるインデックス	221
	インデックス更新から見る RDB との違い	221
	インデックス更新が非同期であることの影響範囲	222
	READ YOUR OWN DATA (自分自身のデータを読む)	223
	永続化や複製の場合と比べた、インデックスの特殊性	224
	一貫性を保証するためのオプション	224
15.5	サンプルアプリケーション紹介	225
15.6	CEAN スタック	225
15.7	アプリケーション概要	226
	アプリケーション利用方法	227
	プログラム解説	229
15.8	ODM(オブジェクトデータモデラー): Ottoman.js	231
付録 A	NoSQL 性能評価: MongoDB, Cassandra との比較	233
A.1	ベンチマーク概要	234
	評価基準・ツール	234
	ワークロード・カテゴリー	234
	環境	235
	共通条件	236
	データベース固有の構成	237
A.2	ワークロード別詳細	238

目次

ワークロード A: 更新処理	238
ワークロード E: レンジスキャン	239
クエリ 1 : ページネーション (OFFSET と LIMIT による フィルター)	242
クエリ 2 : JOIN (テーブル結合)	243

第 1 章

プロログ：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、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、全然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よいですか？」

「自分でよければ。私が LT で話した内容に関係してますか？」

「はい。といっても、かなり漠然とした相談になってしまうのですが、実は先ほど発表したアプリケーションのプロトタイプを作るに当たって、どのデータベースを使おうか悩んでいます。データベースについて、広い知見をお持ちのようでしたので。」

「広いかどうかは分からないけど。どんな部分で悩んでいますか？ 右も左も分からないという感じではないですよね。」

「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

第1章 プロローグ：データベース選択を巡る対話

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、データ構造については試行錯誤すると思うんです。それだけなら、都度、テーブル定義を修正しながら進めても良いのですが...」

「ふんふん」

「プロトタイプとは言っても、ドメインエンティティが素直に反映された設計の、可読性と保守性の高いコードを維持したいんです。これはラストで！」

「分かります。」

「それで、ORM(オブジェクトリレーショナルマッピング)についても考え始めると、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、既に何か RDB 以外で検討しているのじゃないかと思うんですが、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 ドキュメント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のではないかと思ったんですが...」

「そうですね。だけど？」

「それはそれで実際に実装を始めようとする、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...」

「そうですね、データ構造は JSON として一般性があるけれど、プログラミングインターフェイスについては、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？」

「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、現時

第1章 プロローグ：データベース選択を巡る対話

点では、特定のクラウドベンダーに縛られるような形は取るつもりはなく、開発中は、コンテナは使うかもしれませんが、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...Couchbase って聞いたことがありますか？」

「う〜ん、正直名前だけは聞いたことがあるような無いような...Apache プロジェクトでしたっけ？」

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

「それで混乱したのかもしれませんが。Couchbase については何も知らないものとして、教えていただけますか？」

「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...」

「はい、それは私の希望しているところでもあります。」

「あとは、ご相談を聞いて、Couchbase のことを考えたのは、SQL を使えるところ...」

「そこをもう少し詳しく教えてもらっても良いですか？ 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が... あるいは逆の形で、昔からある 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。」

「SQL を JSON 用に拡張する時の一つの形としては 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 のシンタックスが保持される代わりに、開発者が期待していたもののギャップが生じる面があるのではないかと思います。」

「はい。」

第1章 プロローグ：データベース選択を巡る対話

「一方で、関数の追加というレベルでなく、SQL のシンタックス自体を JSON データ構造へのクエリに対して拡張するというアプローチがあります。」

「対照的ですね。」

「JSON のデータ構造として、例えば、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。」

「これまで、期待していたものと違う、とかギャップが生じるとか、具体的なことを言わずに進めてきましたが、何が一番、ギャップとして大きいと感じていますか？」

「そうですね。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…」

「意地悪な質問だったかもしれませんね。実際、自分にとっても、直感的なものではあるのですが、データ構造云々というよりも、端的に言って、データを結合できるかどうか、の部分に開発者にとっての期待値とのギャップがあるんじゃないかと思うんですよね。JSON の構造という話から離れるようにも思うかもしれませんが」

「いえいえ、何となく分かるような気がします。」

「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一つのドキュメントとして保存するのは現実的でないですね。それに、一意のキーでデータ間の関係を表現するのは RDB のお家芸かもしれませんが、JSON 含め、他のデータ構造でも、なんの問題もなく表現可能ですよね。技術的に結合してデータを取り出せるかかどうかが問題となるだけで。」

「確かに。」

「その点、Couchbase の N1QL(ニッケル) は、データ設計者でもあるようなアプリケーション開発者が、期待していることを実現していると思います。」

「ニッケル、ですか？」

「Couchbase のクエリの呼称です。Non-1st-normalization-form Query

第1章 プロローグ：データベース選択を巡る対話

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強要されないから、非第一正規化形クエリ言語、というわけですね。」

「そうそう。今、強要されない、と言われたように、逆に第一正規化しても何の問題もない、というのが結構重要だと思います。」

「確かに。それもクエリの際に結合がどの程度実現されているかどうか、という部分に掛かってくるということですね。」

「そうです。例えば、既存のリレーショナルデータベースで扱っているのと完全に同じ構造のデータを JSON データとして Couchbase で管理した上で、SQL と全く同じクエリを使うといったことも可能です。」

「何となく、技術的に偏った方向に振れすぎる、というような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 に移行するというようなことも、そんなに非現実的ではないと思いますね。メリットがあるかどうかは別として。」

「そういうことなんです。実際にそうするかどうかということだけでなく、そういうことも考えられるくらい、標準的な知識の上に成り立っているかどうか、というのがポイントだと思っていました。」

「標準という意味では、SQL++ という半構造化データへのクエリに関する標準規格があって、N1QL もそれを意識して、進化しています。Apache プロジェクトにも AsterixDB という SQL++ を採用しているデータベースがあります。」

「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、こんな所ですかね。お役に立てたでしょうか？」

「はい、ありがとうございます。」

「あとは、Couchbase を使った NODE.js のサンプルアプリケーションがありますが、見てみますか？」

「是非みてみたいです。」

第1章 プロログ：データベース選択を巡る対話

「ちょっと待ってください。（ラップトップを開いて、Node.js アプリケーションを起動する）画面はこんな感じです。単純なユーザー管理のアプリケーションです。」

「これくらい簡単な方がすぐにイメージし易いです。」

「これがソースです。」

「ユーザーのリスト表示で使っているクエリがこれですね。確かに SQL ですね。」

「登録されているユーザーの表示の他は、ユーザーの追加と削除だけしか実装していませんが、それはこの辺りです。」

「ふうん。INSERT や DELETE のクエリを使うわけではないんですね。」

「N1QL で、それらを使うこともできるけどね。一意のキーで特定されるデータを扱うだけなら、わざわざクエリを文字列で渡して、サーバでそれを解析して実行計画を作成する必要はないということです。」

「RDB っぽくはないけど、ORM を使っている時と似ているかもしれない。」

「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ラッパーを介しているわけではないので、リソース消費の上でも、知識習得の面でも、余計なオーバーヘッドを避けることができている、と言えるんじゃないかな。」

「本当ですね。」

「反対に、検索条件を指定してデータを取得する場合は、SQL 表現の方が開発者にとって直感的なのにも関わらず、ORM を使うために、いわばアンビバレンツな状況に見舞われる、ということも言えるんじゃないかと思います。」

「分かる気がします。私も、そんな感じで、RDB と ORM で進めることに二の足を踏んでいたのかもしれませんが。コード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、他に特別なライブラリを使っているのでもないということですよね？」

「そうですね。必要なのは Couchbase を習得するだけで、その基本は、今

第1章 プロローグ：データベース選択を巡る対話

見てきた N1QL と、幾つかの直接のデータ操作のための API ということで
すね。」

「これ以上、複雑なことをする場合も、基本的に N1QL での操作になって、
SQL が分かっているれば、大丈夫そうですね。」

「大丈夫そうですか？」

「はい、いい感じです。シンプルなのが、良いですね。」

「そうだね。これぐらいシンプルだと、さっきも言ったけど、本格的な開
発になった時に、RDB や他のデータベースを使うことにしても、手戻りは
そんなにない、と言えるかもしれないね。」

「それは、本格的な開発では、そのまま使い続けられない方が良いということ
でしょうか？」

「いやいや、あくまでプログラム設計としてはできるだけ特定の技術に依
存しないよう、抽象化されているのが良い、という話の流れでのコメント
ですよ。RDB よりも後続の技術として、他の NoSQL や NewSQL 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の利点もある
し、性能面で、ディスクの性能がボトルネックにならないメモリを有効活用
したアーキテクチャーが採用されていたり、そのまま本番サービスで活用す
る場合のアドバンテージも大きいと思いますね。」

「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？」

「全然！ Couchbase の良いところは、1 台で全てのサービスを動かしても、
複数台で分散構成をとっても、論理的には変わらないというところです。つ
まり、開発用にラップトップで動かしていても、スタンバイモードと言った
ような特殊なモードで動いているわけではないので、データベースをより本
格的な環境に移行した後でも、アプリケーション側をなんら変更する必要が
ないということですね。付け加えるなら、インストールとセットアップも驚
くほど簡単です。」

「それを聞いて、やってみようというハードルが低くなりました。」

「よかったです。他に気になるところありますか？」

第1章 プロローグ：データベース選択を巡る対話

「大丈夫です。とりあえず、触ってみて、それからかな。」

「そうですね。それじゃあ、私からも良いですか？」

「何でしょう？」

「発表されていたサービスについて、聞いてみたいことがあるんですが...」

「何でも聞いてください！」

会話はさらに続くようです。それぞれの関心を交差させながら...

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、大きくいて、次の二つがあります。

- 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)*¹です。Couchbase Mobile は、Couchbase Lite と Sync Gateway という二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)*²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)*³です。

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

*² <https://developer.couchbase.com/open-source-projects/>

*³ <https://developer.couchbase.com/open-source-projects/>

■コラム:

本書では、題名にある通り、Couchbase Server を扱います。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。その意味でも明らかな通り、Couchbase Server と Couchbase Lite とは、それぞれ明らかに異なるプロダクト/テクノロジーであり、「ドキュメントデータベース」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、そのまま 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。

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、二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、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:

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に固有の機能についてもコミュニティエディションとの差分を示す目的にて紹介しています。

2.2 関連データベース技術

Couchbase Server を理解するための前提知識として、以下の用語について把握しておくとい良いでしょう。

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

ドキュメントデータベース

ドキュメント指向データベース、ドキュメントストアとも呼ばれます。

参考情報

Wikipedia Document-oriented database^{*5}

キーバリューストア

キーバリューストアとも呼ばれます。また、KVS という略称も広く用いられています。

参考情報

Wikipedia Key-value database^{*6}

インメモリデータベース

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。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）を用いた永続性と一貫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る。あらかじめ永続化されているデータの純粋なキャッシュとして用いるのではなく、動的に更新されるデータベースとしてインメモリデータベースを用いる際には、メモリ上のキューを介して外部でデータを永続化することによって、事後的な一貫性を実現しながら、永続化装置の性能に影響されずに、インメモリデータベース本来の応答性能を担保するアーキテクチャーが用いられるのが一般的。

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

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

参考情報

Wikipedia In-memory database^{*7}

リレーショナルデータベース

日本語では、関係データベース、と呼ばれます。関係モデル（リレーショナルデータモデル）に基づいて、データを扱います。

参考情報

Wikipedia Relational database^{*8}

クエリ

データベースにおけるクエリは、データベースへの問合せ（Query）として、普通名詞化されています。関係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(Standard Query Language) 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）のみではなく、追加、削除、更新を含んでいます。

参考情報

Wikipedia Query^{*9}

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。

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

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

^{*9} <https://en.wikipedia.org/wiki/Query>

参考情報

Wikipedia Database transaction^{*10}

^{*10} https://en.wikipedia.org/wiki/Database_transaction

第 3 章

Couchbase Server とは何か？

3.1 Couchbase Server の歴史

Couchbase, Inc. の設立

Couchbase, Inc. は、2011 年に、Membase 社と CouchOne 社の合併により設立されました。Membase 社は、memcached プロジェクト^{*1}のリーダー等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 プロジェクト^{*2}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。

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

ドキュメント指向データベースである Couchbase Server は、JSON デー

^{*1} <https://memcached.org/>

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

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

データを扱うために必要な機能を提供している一方、キーバリューストアとしての側面を持ち、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が、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キテクチャーが採用されており、データ操作（リード・ライト）において、ミリ秒以下の応答性能を実現しています。

上に素描した Couchbase Server のコア機能は、Data サービスと呼ばれます。

■コラム:

CouchbaseServer は、複数の種類の Couchbase サービスからなります。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。

Couchbase Server の発展

クエリ

Couchbase Server 4.0 にて、N1QL(Non 1-st Normalization 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、日本語で非第一正規化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 によるクエリは、Query サービスと Index サービスで実現されます。

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

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。全文検索機能は、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うアプリケーションに検索機能を実装するために使うことのできる機能が備わっています。

Couchbase Server 6.6 にて、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:

本書では、Search サービスについては扱いません。

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、イベントーコンディションーアクション・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。

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

■コラム:

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。

分析

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(ハイブリッドトランザクション・アナリティカル・プロセッシング) 環境として利用可能になりました。OLTP 環境と OLAP 環境を統合するために ETL 処理を実装する必要はありません。また、分析用にデータのスキーマを変更する必要はありません。Couchbase Server をバックエンドデータベースとして利用しているアプリケーションのパフォーマンスを妨げることなく、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です。

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ています。また、MPP (マッシュ・パラレル・プロセッシング) 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 サービスが分析対象として扱う範囲は、従来 Analytics サービスがその一部であるクラスター内部のデータのみでしたが、Couchbase Server 6.6 からは、外部の Couchbase Server クラスターのデータ、および S3 ベースのデータにまで拡張されました。

外部の BI ツールから、Analytics サービスへアクセスするための

第3章 Couchbase Server とは何か？ 3.2 NoSQL と現在の技術動向

ODBC/JDBC ドライバーが³、CDATA³社から提供されています。

■コラム:

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。

参考情報

Couchbase 公式サイト About Us⁴

Couchbase ブログ Couchbase Server Editions Explained⁵

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 と SoE

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』⁶では、以下のように定義されています。

³ <https://www.cdata.com/drivers/couchbase/>

⁴ <https://www.couchbase.com/about>

⁵ <https://blog.couchbase.com/couchbase-server-editions-explained-open-source-community-edition-and-enterprise-edition/>

⁶ <https://www.meti.go.jp/press/2019/07/20190731003/20190731003-1.pdf>

第3章 Couchbase Server とは何か？ 3.2 NoSQL と現在の技術動向

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、企業文化・風土を変革し競争上の優位性を確立すること

ここで、トランスフォーメーション、つまり変革の対象とされているものは二つあります。一つは、企業が生み出し、顧客に提供する製品やサービス、そしてもう一つは、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のでしょうか、つまり、これまでも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、

第3章 Couchbase Server とは何か？ 3.2 NoSQL と現在の技術動向

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（あるいは、極端な断言かもしれませんが、一つの見方として受け止めていただければと思います）。

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、というものを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。

第3章 Couchbase Server とは何か？ 3.2 NoSQL と現在の技術動向

- 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として、以下の3点を挙げたいと思います。

- **ユーザーの規模感** 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。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。それに伴って、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、ということがいえます。

- **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 (ただし、そのスケールに対する柔軟性については、それぞれ違いがあります)。
- **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

第3章 Couchbase Server とは何か？ 3.2 NoSQL と現在の技術動向

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のできる、という他のデータベースにはない利点があります。

- **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。

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実現しており、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兼ね備えている、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・データプラットフォームとして用いるのに相応しいといえます。

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

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。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、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、全て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もあります。

「インメモリ」というキーワードで、端的に連想されるのは、あるいはパフォーマンスということになるかもしれませんが、ここではより具体的に、応答性能に焦点を当てて検討してみたいと思います。また、「データをメモリーに格納」という表現のみでは、訴求内容が十分明確でないかもしれないという点を考慮して、「永続化装置の性能がボトルネックにならない」という観点を強調したいと思います。

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。

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

リード・スルー

キャッシュ層が、アプリケーションにとってのインターフェイスとして、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、アプリケーションにとってのインターフェイスとして、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のインターフェイスとなるのは、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、データベースとの関係を非同期とする。このことで、同一エントリ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排除く、などの利点が生じる。

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。

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

第 3 章 Couchbase Server とは何か？ インメモリファーストアーキテクチャー

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) と呼ばれます。

インメモリ DB

Oracle Coherence

Oracle Coherence^{*7}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。

Redis

Redis^{*8}を MongoDB と連携する場合には、cdata 社の CData Sync^{*9}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」といったのは、このようなキャッシュ

^{*7} <https://coherence.community/>

^{*8} <https://redis.io/>

^{*9} <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

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二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)。

この場合、内部的な実装は、いわば「ライト・ビハインド」のパターンに相当しますが、一貫性 (Durability) 要件に応じ、ディスク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)。

以下に、以上説明したアーキテクチャーのイメージを示します (画像は、Couchbase Under the Hood: An Architectural Overview^{*10}より引用)。

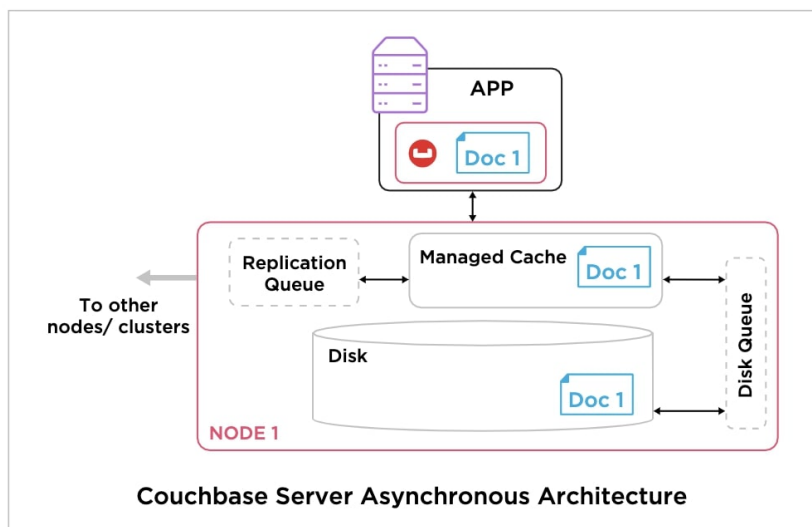


図 3.1: memory first arch

^{*10} https://resources.couchbase.com/c/server-arc-overview?x=V3nd_e

■コラム:

CouchbaseServer は、永続化要件のないデータの扱い についても、サポートしています。これは、「バケット」と呼ばれるデータ格納単位の作成時に Ephemeral（一時的）バケットとして作成することによって実現されます（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）。

3.4 JSON データベース

ここでは、なぜ JSON（データベース）が重要なのかについて整理します。

JSON、その歴史と関連技術

類似技術との比較

まず、先行する技術である、XML との比較から、JSON の特徴を解き明かしていきたいと思います。

■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、配列）のペアの集まり（オブジェクト）によって、ユーザ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。名称と構文は JavaScript におけるオブジェクトの表記法に由来。

XML と JSON の共通点

- データ記述言語（データ記述言語は、マークアップ言語を包含する概念）として、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
- 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

JSON のメリット

歴史的に、JSON は、後発ながら、XML が利用されていたケースを置き換える形で広まってきました。XML に対する JSON のメリットとしては、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 に由来しており、辞書 (マップ) や配列・リストのような多くのプログラミング言語がネイティブに持つデータ構造へ直接コンバートすることができる

JSON とモダンな開発・サービス形態との関係

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、特に重要だったのが、WEB ブラウザー上での扱いだったといえます。WEB ブラウザー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) 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(シングルページアプリケーション)、PWA(Progressive web application) 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 (SPA, PWA) がなぜ重要かと言えば、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。

さらには、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、JSON は、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 に遥かに勝る）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。

参考情報

Wikipedia Ajax^{*11}

Wikipedia SPA(シングルページアプリケーション)^{*12}

Wikipedia PWA(Progressive webapplication)^{*13}

^{*11} <https://ja.wikipedia.org/wiki/Ajax>

^{*12} <https://ja.wikipedia.org/wiki/%E3%82%B7%E3%83%B3%E3%82%B0%E3%83%AB%E3%83%9A%E3%8>

^{*13} https://en.wikipedia.org/wiki/Progressive_web_application

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^{*14}

3.5 トランザクション今昔

ここでは、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、関心が深いと思われるトピックである「トランザクション」について、概観を整理していきます。

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

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 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-and-set (CAS) API」による楽観的 (オプティミスティック) ロックの二つがあります。

マルチドキュメントトランザクション

複数のレコード間を、外部キーによって関連づけることは、ドキュメント指向データベースであっても、採用しうる設計です。そのことを踏まえ、複数レコード/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、次の二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)

^{*14} <https://digital-marketing.jp/seo/what-is-progressive-web-apps/> 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。

- 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だといえることが言えます。

その一方で、NoSQL テクノロジーが従来想定し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、複数ドキュメントに跨がるトランザクションの実現に対する要望も高くなっていったものと考えられます。

例えば、Couchbase の過去のブログでは、トランザクションを表現するドキュメントを使うことによって、複数ドキュメントに跨がるトランザクションをアプリケーションとして実現する例が紹介されていたりもします。NoSQL データベースの制限をデータベース外部で解決するという試みは、様々な形で行われています（一例として NEC の InfoFrame Relational Store(IRS)*¹⁵があります）。

NoSQL に、マルチドキュメント・トランザクションが、データベースの機能として実装された背景には、ドキュメント指向データベースのユースケースの範囲が、大きく変わってきていることが示唆されていると考えられます。

Couchbase のマルチ・ドキュメント・トランザクションは何であるか

Couchbase のマルチ・ドキュメント・トランザクションは、あるバケットに含まれる複数ドキュメントに跨がることができます（さらには、異なるバケットに跨がることさえできます）。一つの論理バケットは、物理的に複数ノードに跨がるので、トランザクションは、当然複数ノードに跨がることになります（NoSQL の世界でトランザクションの実現という時、分散アーキ

*¹⁵ https://jpn.nec.com/infoframe/relationalstore/docs/irs_couchbase_live_tokyo_2015.pdf

テクチャーとしてのクラスター内の複数のノード間のトランザクションという観点から語られることがあるため、注釈しておきます)。

Couchbase のマルチ・ドキュメント・トランザクションは何でないか

ここで注意しなければならないのは、Couchbase(NoSQL) 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、ここでは、上で触れた単一ドキュメント更新時のように、悲観的ロックを選択することはできないことです。

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、非常に興味深いものだと考えます。以下の図をここで引用することは、その意味で、意義深いと思われます。

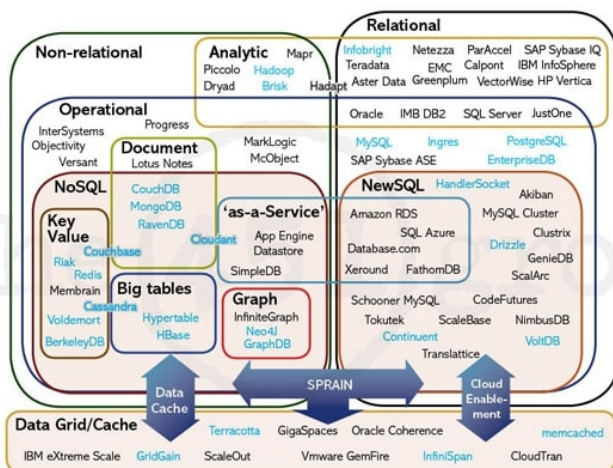


図 3.2: newsql

画像は、NoSQL, NewSQL and Beyond^{*16} 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、端的に言えば、Google Spanner^{*17}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ー（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、Google Spanner については、「Fully managed relational database with unlimited scale, **strong consistency**, and up to 99.999% availability.」と謳われています。クローンごとに若干打ち出し方が異なっているのは、CockroachDB^{*18} 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^{*19} は、「distributed SQL database for elastic scale and **real-time analytics**」と定義しています。YugaByteDB^{*20} 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 が、後天的に獲得した複数レコード/ドキュメント間のロックという特徴が、リードコミテッドのトランザクションレベルに基づいている一方、NewSQL のロックは、より厳格であること（シリアライザブルのトランザクションレベル）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 を「(New)SQL/RDB」として位置付けてい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設計ということになります。

^{*16} <https://www.infoq.com/news/2011/04/newsql/>

^{*17} <https://cloud.google.com/spanner>

^{*18} <https://www.cockroachlabs.com/>

^{*19} <https://pingcap.com/>

^{*20} <https://www.yugabyte.com/>

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 RDB 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

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さえないため、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、それ自体単体で見た場合悲観的ロックと比較して軽量の処理であるはずの、楽観的ロックの実装を利用した場合、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内容ではなく、コストや、アーキテクチャ的なアドバンテージなども考慮された上で、判断が必要になるところだと思われます。

参考情報

Couchbase 公式ドキュメント Transactions^{*21}

Couchbase ブログ Optimistic vs Pessimistic Locking -- Which One Should You Pick?^{*22}

Couchbase ブログ Couchbase Brings Distributed Multi-Document

^{*21} <https://docs.couchbase.com/server/current/learn/data/transactions.html>

^{*22} <https://blog.couchbase.com/optimistic-or-pessimistic-locking-which-one-should-you-pick/>

ACID Transactions to NoSQL^{*23}

Couchbase ブログ Understanding Distributed Multi-document ACID Transactions in Couchbase^{*24}

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。「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、リレーショナル・構造化データベースに対比して、「準構造化データベース」とでも呼んだ方が、技術的には、実態をよりよく表現することができるといえます（あるいは、いっそ JSON データベースと読んだ方が、実態が明確になるでしょう）。

「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(構造化) イコール、リレーショ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、と解釈できます。「構造化」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、テーブル形式でないデータも、紛れもなく「構造化」を持っているといえます。

テーブル形式は、リレーショナルデータベースにデータを格納する際の制約である、第一正規化を強制された構造化、ということができ、その意味では、「準構造化データ」というよりも、「非第一正規化形データ」とでも呼ぶ方が、

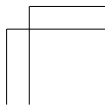
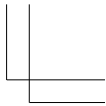
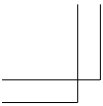
^{*23} <https://blog.couchbase.com/couchbase-brings-distributed-multi-document-acid-transactions-to-nosql/>

^{*24} <https://blog.couchbase.com/distributed-multi-document-acid-transactions-in-couchbase/>

実態を表しているのではないのでしょうか？（ここでの疑問形は、流通している用語を変更することを訴えるためではなく、そうした地点から考えてみることを示唆する目的から使われています）

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-normalization-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 固有の要素が存在しない場合、RDB で使われているものと同じ SQL を使うことができるといえます。そうした面から、Couchbase Server は RDB のスーパーセットである、ということさえ言えるかもしれません（データ形の種類について、JSON は SQL 標準よりも限定されています。また、トランザクションについても、RDB よりも限定的であることは忘れてはなりません）。依然として存在する様々な違いに関わらず、このような表現をした意図としては、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して目されている、という背景に対して、行き過ぎを正す意図があります。多くの NoSQL が、RDB と異なる（ビッグデータや分析等の）ユースケースで使われることを前提としているのに対して、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 のユースケース（の大部分）と共通であるといえます。



第 4 章

Couchbase Server を使ってみる

4.1 環境構築手順

インストーラー利用

ダウンロード

Couchbase Server のインストーラーは、<https://www.couchbase.com/downloads> からダウンロードすることが可能です。

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。

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。

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

Docker 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 couchbase
```

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。

`http://localhost:8091` にアクセスします。

下記のような画面が表示されます。

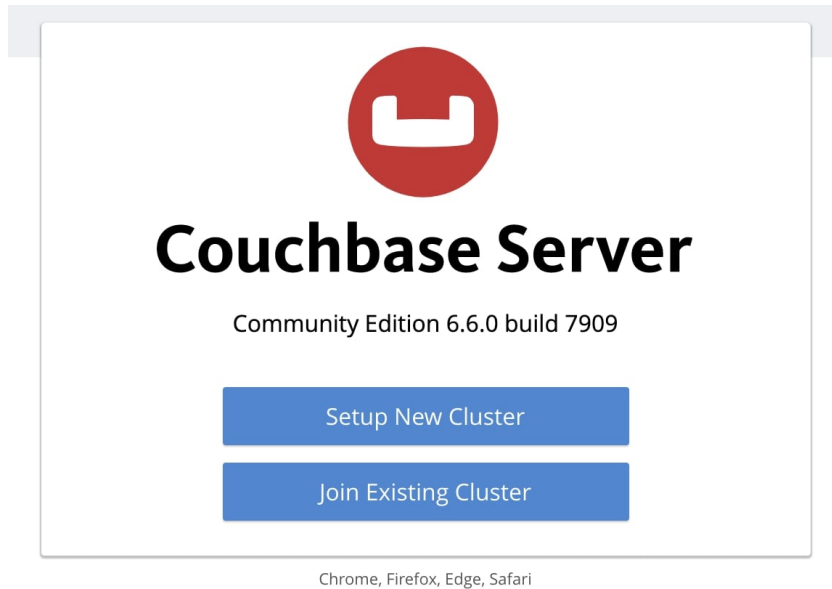


図 4.1: initial

ログの確認

下記以降のコマンドで用いている `cb` は、`docker run` コマンドの `--name` オプションで指定したものです。

```
$ docker logs cb
```

Docker イメージの確認

```
$ docker images
```

プロセスの確認

```
$ docker ps
```

上記の手順により couchbaseREPOSITORY の community という TAG 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・起動

コンテナをストップします。

```
$ docker stop cb
```

実行中のコンテナを確認します。下記のコマンドにより、結果が表示されず、ブラウザで WEB コンソールにアクセスできないことも確認します。

```
$ docker ps
```

実行されていないコンテナを確認するには、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 `cbbbackup` 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/cbbbackup -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をします）。

```
$ 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
```

クラスター初回構築

■ コラム:

クラスターの構築、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 管理コンソール を使った手順について解説します。

WEB 管理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、下記の様な画面が表示されます。

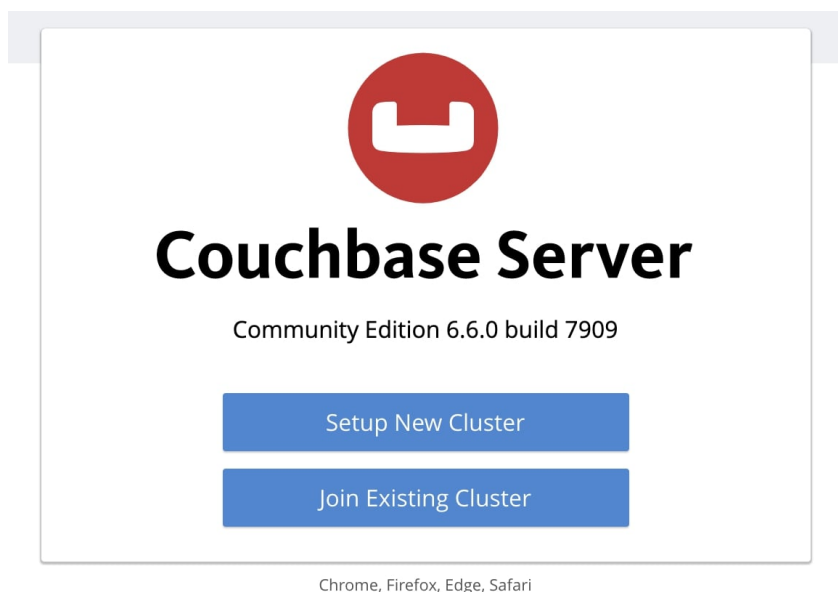
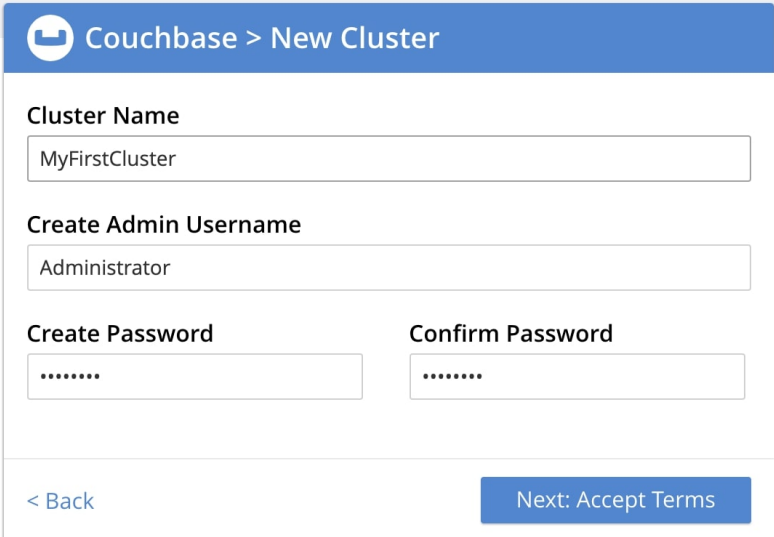


図 4.2: initial

新規クラスターセットアップ

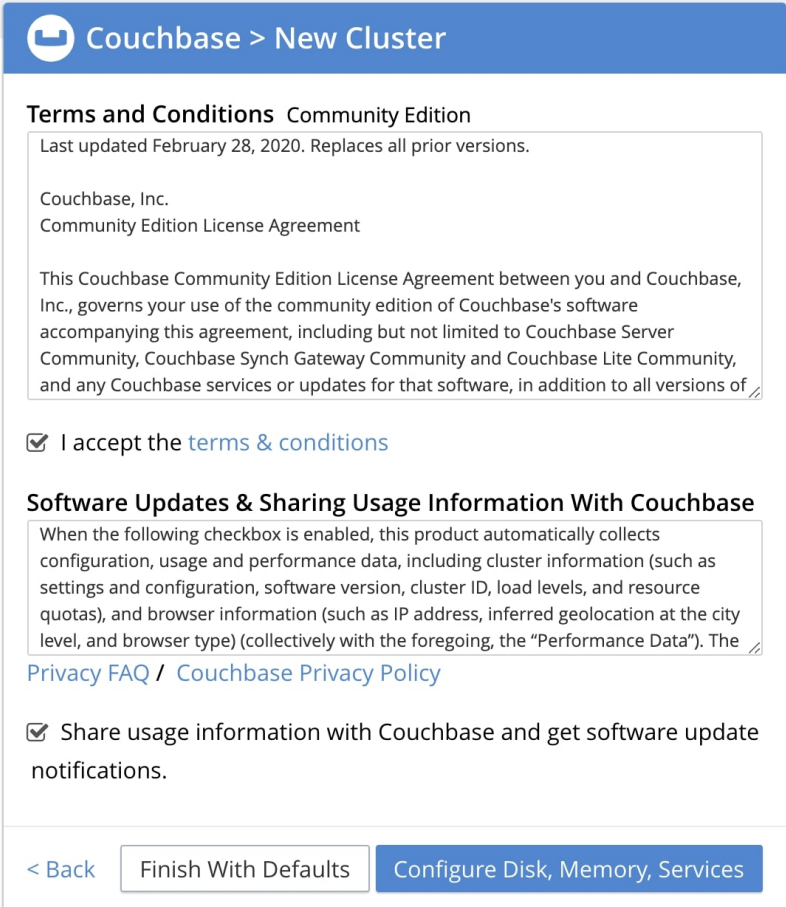
1. 上記の画面に見える通り、インストール直後に WEB コンソールにアクセスした際の初めの選択肢は、「Setup New Cluster」と、「Join Existing Cluster」の二つです。ここでは、前者を選択します。
2. 次の様な画面が表示されるので、適宜入力します。



The screenshot shows the 'Couchbase > New Cluster' configuration page. It includes a blue header with the Couchbase logo and title. Below the header, there are four input fields: 'Cluster Name' with the value 'MyFirstCluster', 'Create Admin Username' with the value 'Administrator', 'Create Password' with masked characters '*****', and 'Confirm Password' with masked characters '*****'. At the bottom left is a '< Back' link, and at the bottom right is a blue button labeled 'Next: Accept Terms'.

図 4.3: new cluster

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



Couchbase > New Cluster

Terms and Conditions Community Edition

Last updated February 28, 2020. Replaces all prior versions.

Couchbase, Inc.
Community Edition License Agreement

This Couchbase Community Edition License Agreement between you and Couchbase, Inc., governs your use of the community edition of Couchbase's software accompanying this agreement, including but not limited to Couchbase Server Community, Couchbase Synch Gateway Community and Couchbase Lite Community, and any Couchbase services or updates for that software, in addition to all versions of

☒ I accept the [terms & conditions](#)

Software Updates & Sharing Usage Information With Couchbase

When the following checkbox is enabled, this product automatically collects configuration, usage and performance data, including cluster information (such as settings and configuration, software version, cluster ID, load levels, and resource quotas), and browser information (such as IP address, inferred geolocation at the city level, and browser type) (collectively with the foregoing, the "Performance Data"). The

[Privacy FAQ](#) / [Couchbase Privacy Policy](#)

☒ Share usage information with Couchbase and get software update notifications.

< Back Finish With Defaults **Configure Disk, Memory, Services**

図 4.4: terms and conditions

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

クラスター設定

Couchbase > New Cluster > Configure

Host Name / IP Address Fully-qualified domain name
127.0.0.1

Data Disk Path Path cannot be changed after setup
/opt/couchbase/var/lib/couchbase/data
Free: 42 GB

Indexes Disk Path Path cannot be changed after setup
/opt/couchbase/var/lib/couchbase/data
Free: 42 GB

Service Memory Quotas Per service / per node

<input checked="" type="checkbox"/> Data	311	MB
<input checked="" type="checkbox"/> Query	-----	
<input checked="" type="checkbox"/> Index	512	MB
<input type="checkbox"/> Search	256	MB

TOTAL QUOTA 823 MB

[< Back](#) [Save & Finish](#)

図 4.5: cluster configure top

The screenshot shows the 'Couchbase > New Cluster > Configure' page. At the top, there's a text input field containing '/opt/couchbase/var/lib/couchbase/data' and a note 'Free: 42 GB'. Below this is the 'Service Memory Quotas' section, which is 'Per service / per node'. It lists four services: Data (checked, 311 MB), Query (checked, -----), Index (checked, 512 MB), and Search (unchecked, 256 MB). A 'TOTAL QUOTA' of 823 MB is shown. A grey box indicates 'RAM Available 1989MB' and 'Max Allowed Quota 1591MB'. The 'Index Storage Setting' section has two options: 'Standard Global Secondary' (selected) and 'Memory-Optimized' (with an info icon). At the bottom, there are '< Back' and 'Save & Finish' buttons.

Service	Quota (MB)
<input checked="" type="checkbox"/> Data	311
<input checked="" type="checkbox"/> Query	-----
<input checked="" type="checkbox"/> Index	512
<input type="checkbox"/> Search	256

TOTAL QUOTA 823 MB

RAM Available 1989MB Max Allowed Quota 1591MB

Index Storage Setting

- ☒ Standard Global Secondary
- ☐ Memory-Optimized ⓘ

図 4.6: cluster configure bottom

1. 不要なサービスのチェックボックスを外すことにより、このノードでは、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、クラスターからの削除、再追加が必要）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。上の画面の例では、Search（全文検索）を無効にしています。

2. 「Save & Finish」を押下し、終了します。最終的にこの様な画面が表示されます。

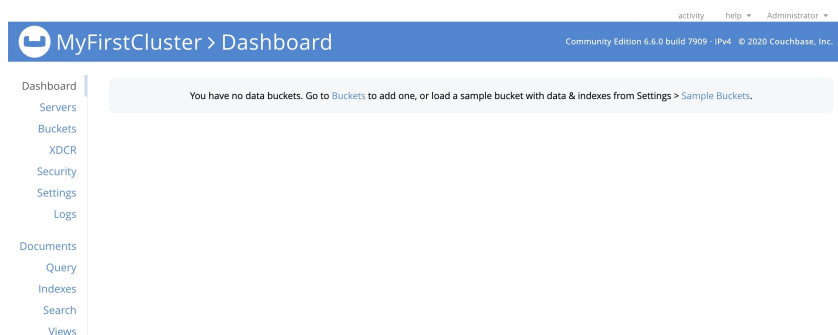


図 4.7: dashboard initial

ここまでの手順を経て、1 ノードで構成されるクラスターとして Couchbase Server が構築されたことになります。

クラスター構成変更

■コラム:

クラスターの構築、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 管理コンソールを使った手順について解説します。

既存のクラスターにノードを追加あるいは削除することによって、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。

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 管理コンソール）から実行
- クラスターへ追加するノード（の WEB 管理コンソール）から実行

■クラスターの既存ノードから実行

1. サイドメニューから、「Servers」を選択します。
2. トップバーから、「ADD Server」を選択します。
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」を押下します。

■クラスターへ追加するノードから実行 インストール直後の状態では、下記のような画面が表示されます。

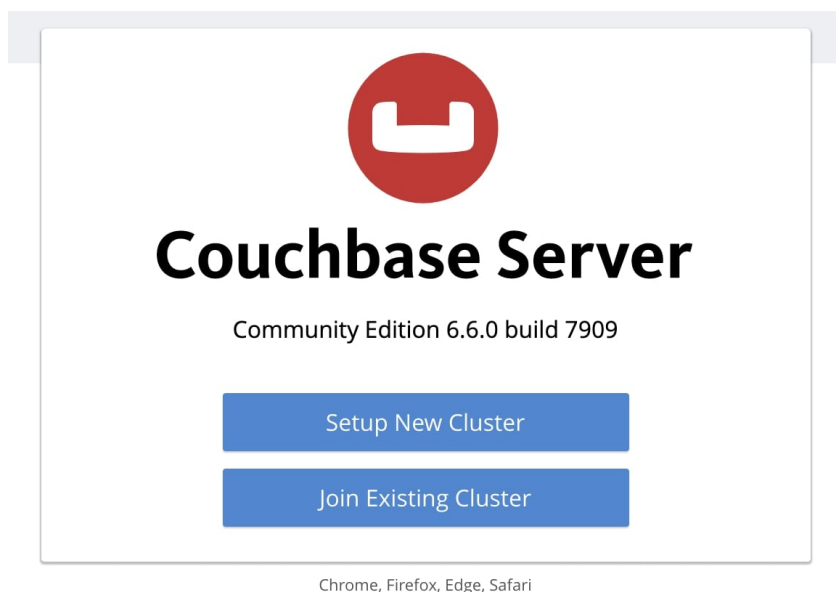


図 4.8: initial

1. 「Join Existing Cluster」を押下します。
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」を展開し、編集します（ここでは説明を割愛します）。
4. 「Join With Default Configuration」を押下します。

クラスターがバケットを持っていない場合には問題ありませんが、既にそのクラスターにバケットがあり、バケットにデータが格納されている場合には、追加したノードを含めてデータを均等に分散する必要があります。そのためには、Servers 画面右上の「Rebalance」ボタンを押下します。

第 4 章 Couchbase Server を使ってみる 4.2 管理コンソールの使い方

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、拡大します。
3. 右下の「Remove」ボタンを押下します。

クラスターがバケットを持っていない場合には問題ありませんが、既にそのクラスターにバケットがあり、バケットにデータが格納されている場合には、ノードをクラスターから取り外した直後の状態では、アクティブコピーとレプリカコピーがそれぞれ一部分失われており適切な状態ではありません。この問題を解消するために Servers 画面右上の「Rebalance」ボタンを押下します。

4.2 管理コンソールの使い方

バケット作成

1. データの格納の単位である Bucket を作るため、「Bucket」リンクをクリックします。
2. 表示されるのは、バケット一覧画面です。右上の「ADD BUCKET」リンクをクリックします。

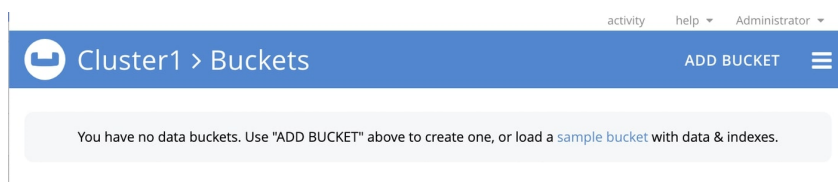


図 4.9: buckets initial

1. ここでは、バケット名として、「MyFirstBucket」を入力しています。

第 4 章 Couchbase Server を使ってみる 4.2 管理コンソールの使い方

2. 「Advanced bucket settings」を展開し、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、今回の手順で用いる 1 ノード構成のクラスターでは、チェックボックスを外さない場合は、バケット作成後、常時ワーニングが表示されてしまいます）
3. 開発時の利便性のため、Flush を有効にしておきます。（Flush ボタンが有効になり、このボタンを押下することで、バケットのデータをすべて削除することが可能になります。）

第 4 章 Couchbase Server を使ってみる 4.2 管理コンソールの使い方

Add Data BucketX

Name

MyFirstBucket

Memory Quota in megabytes per server node

311

MB

other buckets (0 B)

this bucket (311 MB)

remaining (0 B)

Bucket Type

☒ Couchbase ☐ Memcached ☐ Ephemeral

▼ Advanced bucket settings

Replicas

☐ Enable

☐ Replicate view indexes

Ejection Method ⓘ

☒ Value-only ☐ Full

Bucket Priority ⓘ

☒ Default ☐ High

Durability Level ⓘ

None ▼

Auto-Compaction ⓘ

☐ Override the default auto-compaction settings?

Flush ⓘ

☒ Enable

CancelAdd Bucket

図 4.10: add bucket

第 4 章 Couchbase Server を使ってみる 4.2 管理コンソールの使い方

1. 「Add Bucket」 ボタンを押下します。

ユーザー作成

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。
3. 「Add New User」ダイアログで、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、アプリケーションから利用するバケット名を開き、「Application Access」にチェックします。
5. 「Add User」 ボタンを押下します。

データ登録

外部データファイル利用

既存のデータベースからの移行等の目的で、ファイルを介して、データを Couchbase Server へ登録する場合があります。

ここでは、WEB 管理コンソールからの外部ファイルを用いたデータ登録手順を示します。

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 Documents」を選択します。
 3. 左上部の「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。下記の通り、設定を行います。
- Parse File As: ファイル形式 (JSON, CSV)

第4章 Couchbase Server を使ってみる 4.2 管理コンソールの使い方

- Destination Bucket: データを登録するバケット名
 - Import With Document ID: ドキュメントのキーとして利用するフィールド (トグルボタンの「Value of Field」を選択の上、リストから選択)
1. ここまでの入力した内容に従って、左上部画面の `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が可能です。
 2. 画面下の「Import Data」ボタンを押下します。

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おり、これを使って、Couchbase Server をどのように利用するか、理解を深めることができます。

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。

クエリ実行

1. サイドメニューから、「Query」を選択します。
2. 上部メニューから、「Query Workbench」を選択します (デフォルトで表示されていない場合)。
3. 左上部の「Query Editor」に N1QL を入力します。
4. 「Execute」ボタンを押下します。

インデックス作成

実行

1. サイドメニューから、「Query」を選択します。
2. 上部メニューから、「Query Workbench」を選択します（デフォルトで表示されていない場合）。
3. 左上部の「Query Editor」に、N1QL を用いて、CREATE INDEX DDL を入力します。
4. 「Execute」 ボタンを押下します。

確認

1. サイドメニューから、「Indexes」を選択します。
2. 上記で作成したインデックスが表示され、それぞれ（列左端の）「status」が「ready」であることを確認します。

■コラム:

N1QL クエリやインデックスの詳細については、別章を参照ください。

第 5 章

データ

5.1 データ管理

ここでは、Couchbase Server におけるデータ管理について解説します。
また、ユーザーにとっての選択肢を具体的に見ていきます。

ドキュメント

Couchbase Server のデータ格納単位は、ドキュメントです。その構造を説明します。

ドキュメントキー（またはドキュメント ID）

- 可変長
- 最大 250 バイト
- 値は、バケット内において一意

メタデータ

- 固定長
- Couchbase バケットに保存されているドキュメントの場合：56 バイト

イト

- Ephemeral バケットに保存されているドキュメントの場合：72 バイト

ドキュメント値

- 可変長 (最大 20MB)
- 通常は JSON であることを想定しているが、バイナリデータの保存が可能

ドキュメント有効期限

ドキュメントには有効期限を設定することができます。

バケット

Couchbase Server ではドキュメントは、論理的コンテナ/キースペース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。

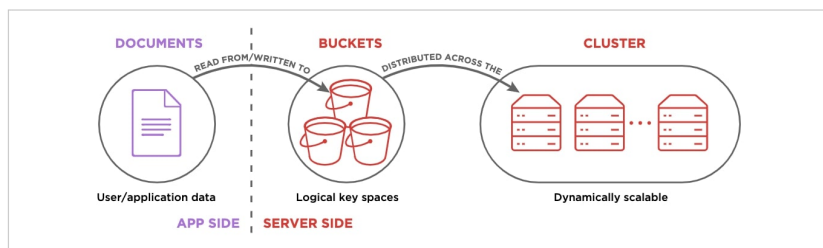


図 5.1: bucket concept

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

■コラム:

バケットに含まれるデータは、物理的には複数のノードに跨って管理されますが、ここでは、ユーザー/アプリケーションからみた「論理的」なレベルのデータ管理について解説しています。

バケット種類

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、ディスクに対して非同期でデータの永続化を行います。
- 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:

CouchbaseServer のバケット種類には、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 バケットは非推奨 (deprecated) とされています。

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

Couchbase Server は、可能な限りすべてのデータ（アクティブ、およびレプリカ含め）をメモリに保持しようとしています。

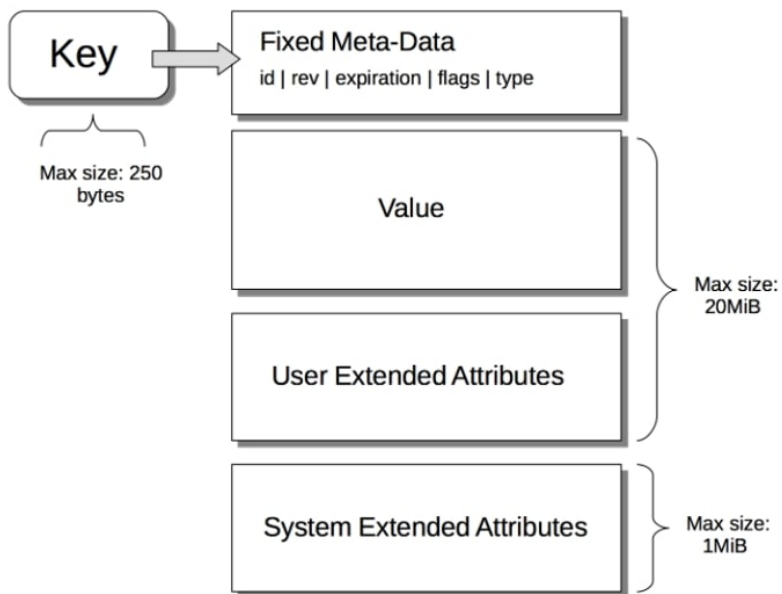


図 5.2: document conceptual diagram

(画像は、Couchbase Server ドキュメント Size Limits^{*2}より引用)

メモリからのデータ排出 (Ejection)

メモリ内のデータがバケットの（ハイ・ウォーターマークと呼ばれる）メモリクォータの 85 %に達した場合、サーバーは、最近使用されていないものの一部をメモリから排出（削除）し始めます。データ排出プロセスは、メ

^{*2} <https://docs.couchbase.com/server/current/learn/data/data.html#size-limits>

メモリ内データがバケットメモリクォータの 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。

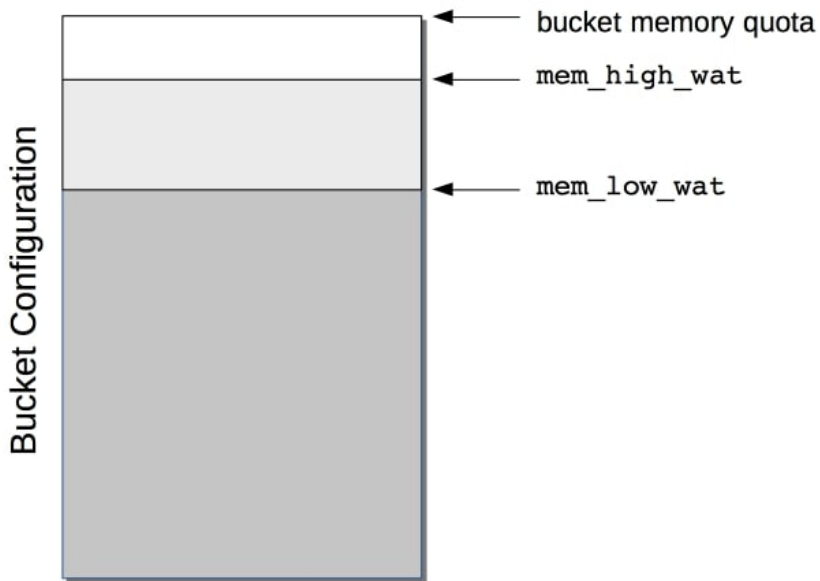


図 5.3: ejection conceptual diagram

(画像は、Couchbase Server ドキュメント Ejection^{*3}より引用)

排出方法 (Ejection Method)

排出方法として、Value-Only と Full の何かを選択可能です。

- **Value-only** 排出の際、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、最高のパフォーマンス

^{*3} <https://docs.couchbase.com/server/current/learn/buckets-memory-and-storage/memory.html#ejection>

スを提供します。

- **Full** 排出の際、すべて（キー、メタデータ、値を含む）が排出されます。メモリ要件を軽減します。

排出方法の選択

以下の二つのデータ保存要件の性格の違いによる、排出方法の選択を行うことが考えられます。

- **データ量は一定であることを想定（増加した過去データを別の場所へ退避する）** この場合には、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、最高のパフォーマンスを実現することができます。
- **データ量は経年的に増加することを想定** この場合、一般的なデータベースの場合、ディスクの容量に注目していれば十分だったとすると、データ数に応じて増えるメタデータがメモリ容量に与えるインパクトについても計画する必要があります。**Full** を選択することで、データ数（メタデータ量）増加によるインパクト、つまり一般的なデータベースの場合考慮する必要がなかった検討要素を回避することができます。

排出方法の設定・反映

排出方法の設定は、バケット作成時に行うのみではなく、作成後の変更も可能です。ただし、この変更は、（後から変更可能なバケット設定の中でも珍しく）、バケットのリスタート（設定変更時に暗黙に行われる）を必要とすることに注意が必要です。

監視と警告

事前の計画が基本である一方、運用中の監視も重要な要素です。

監視

以下は、WEB 管理コンソール上の監視項目のイメージです（同じ内容をコマンドラインや、外部監視システムとの連携で利用することができます）。データ全体のサイズ（上）に加え、メタデータのサイズ（下）を監視することができます。

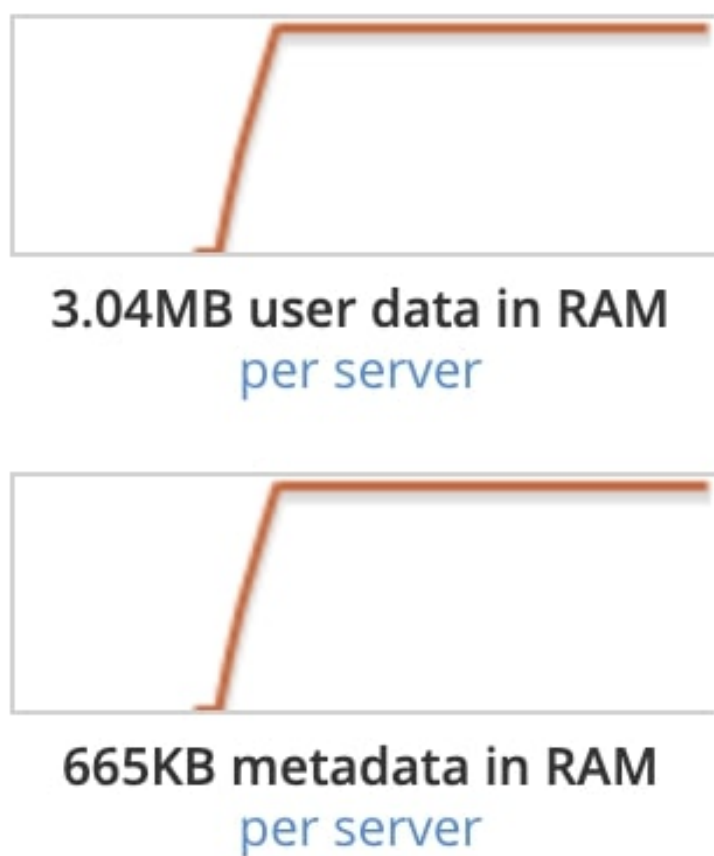


図 5.4: data monitoring

警告

バケットのメモリクォータに対して占めるメタデータサイズの割合の超過に対する警告を利用することができます。

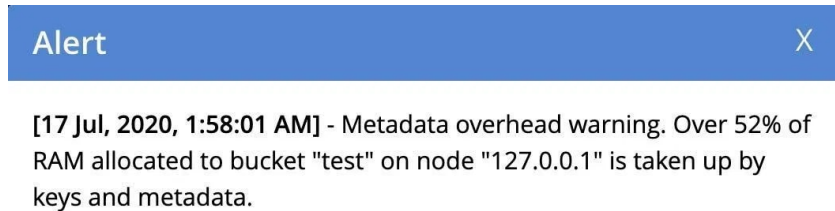


図 5.5: data alert

ドキュメント有効期限

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ことも、バケットレベルで設定することも可能です。両方に設定されている場合、より短い有効期限が採用されます。

■コラム:

バケットレベルの有効期限設定は、エンタープライズエディションでのみ利用可能です。

参考情報

Couchbase ブログ A Tale of Two Ejection Methods: Value-only vs. Full^{*4}

Couchbase ブログ The Best Database For Storing Images Might Not

^{*4} <https://blog.couchbase.com/a-tale-of-two-ejection-methods-value-only-vs-full/>
排出方法にフォーカスしたブログ記事

Be a Database At All^{*5}

5.2 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・整理できます。

- コレクションは、複数のドキュメントをグループ化するための機構です。
- スコープは、複数のコレクションをグループ化するための機構です。

存在意義

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（ビジネスエンティティ）の違いを、コレクションによって表現することができます。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る N1QL クエリでは、コレクションは、リレーショ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（具体的にいえば、FROM 句で用いられます）。これによって、リレーショナルデータベースから Couchbase Server への移行が容易に実現されます（端的にいえば、データ構造を変更しない限り、同じクエリを用いることが

^{*5} <https://blog.couchbase.com/the-best-database-for-storing-images-might-not-be-a-database-at-all/> データベース (Couchbase Server) 利用の目的として、（一般によく取り上げられるものの中でも）必ずしも適切でないユースケースもあることを指摘しながら、Couchbase Server のデータ管理、排出方法メカニズムとの関係を論じています。

できます)。念のため付け加えると、コレクションによって、その中に含まれるドキュメントの構造が一律のものとして規定されるわけではないため、JSON フォーマットの持つ柔軟性が損なわれることはありません。

スコープの存在意義

スコープによって、コレクションをグループ化することによって、ビジネスエンティティのグループを管理することができます。これによって、マルチテナンシーを実現することができます。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。

機能

適用範囲

スコープとコレクションによるデータ管理によって、Couchbase Server の様々な機能において、以下が可能になります。

- スコープとコレクションごとのアクセス権を割り当てることができます。これによって、各ユーザー/アプリケーションが必要なコレクションのみにアクセスできるようにできます。
- コレクションにインデックスを付けるができます（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が利用可能です。

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

基本仕様

- クラスタごとに最大 1000 個のスコープを作成できます。
- 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。
- クラスタごとに最大 1000 のコレクションを作成できます。
- アイテム名 (ドキュメントのキー) 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 管理コンソール
- CLI
- REST API
- N1QL
- SDK

コレクションとドキュメント有効期限管理

コレクションに対して有効期限を設定することができます。これは、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。Couchbase Server における有効期限の詳細については、ここでは触れません。

種類と命名規則

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、その中にはデフォルトのコレクションが与えられます。それぞれには、`_default` という名前が付けられています。

特定のスコープまたはコレクションを指定せずに作成されたドキュメントは、デフォルトのスコープ内のデフォルトのコレクションに保存されます。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。

バージョン間の整合性

(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 からアップグレードすると、既存のすべてのデータ (ドキュメント) がデフォルトのスコープとデフォルトのコレクションに自動的に配置されます。

ユーザー定義スコープ・コレクションの命名

ユーザー定義のスコープとコレクションには、ユーザー定義の名前を割り当てますが、以下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。

参考情報

Couchbase 公式ドキュメント Scope and Collection^{*6}

^{*6} <https://docs.couchbase.com/server/current/learn/data/scopes-and-collections.html>

第 6 章

クエリとインデックス

■コラム:

Couchbase Server では、RDB とは異なり、クエリの実行にあたり対応するインデックスが存在していることが前提となります。この点を踏まえ、ありうべき混乱を避けるため、本章では、まずインデックスについて解説し、その後、クエリについて解説します。

6.1 インデックス

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

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ーインデックス (GSI) と呼ばれています。別に「プライマリインデックス」、「セカンダリーインデックス」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ーインデックス (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ーインデックス (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 (現在も互換性のために維持されています)。この View がデータノードに対して「ローカル」であったことに対して、データノードのローカルティに依存しない、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。「セカンダリー」という形容には、RDB におけるセカンダリー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、というニュアンスで受け止めることができます。

■コラム:

Couchbase Server には、ローカルセカンダリーインデックス (Local Secondary Index) という概念は存在しません。

■コラム: 過去のバージョンとの違い

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 のインデックスとの違い

Couchbase Server の SGI と、RDB とのインデックスの間には、ユーザーが意識しておくべき、下記のような違いがあります。

- Couchbase Server におけるクエリでは、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。

最後の点について、完全に同じ内容のインデックスを別名で、つまり複数、作成することを許していることにどのような意味があるのでしょうか？

Couchbase Server のインデックスは、RDB/SQL の構文を可能な限り踏襲しながら、RDB にはない、ユーザーによる様々な制御の余地を提供して

います。例えば、これは分散アーキテクチャー独自の部分になりますが、インデックス定義時に、ローカリティを指定することが可能です。同じ内容のインデックスを別名で、複数定義できるという特徴と組み合わせて、複数のノードに同じインデックスを格納することによって、クエリの性能を向上するという使い方ができます。

その他の特徴についても、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:

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。

これらは共に、Multi Version Concurrency Control (MVCC, マルチバージョン・コンカレンシー・コントロール) を実装しており、一貫したインデックススキャン結果と高スループットを提供します。

スタンダードグローバルセカンダリインデックスは、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、データの変更に際して、に高速に処理が行われます。

■コラム:

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。

プライマリインデックス

プライマリインデックスは、バケット単位で指定するインデックスです。

■コラム:

Couchbase Server では、ここで説明するプライマリインデックスとは関係なく、ドキュメントキーには一意制約があります。

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ーグローバル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、処理が失敗するという性質があります。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（下記例の場合指定された `airline` コレクション）全体に対して、インデッ

第6章 クエリとインデックス

6.1 インデックス

クスを作成することができるため、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。つまり、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。とはいえ、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。

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

作成されたインデックスの定義（Definition）を確認すると分かりますが、上記の DDL 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。(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、上記の省略形は、プライマリインデックスに特有のものです)

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

したがって当然、次の例に示すように、明示的に、名前を付けて定義することもできます。

```
CREATE PRIMARY INDEX <インデックス名> ON <バケット名>;
```

■コラム:

プライマリインデックスによる検索は、検索条件の内容に関わらず、

対象キースペース（上記例の場合指定された `airline` コレクション）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、ドキュメントキーによる検索の場合も同様です（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）。実際上、プライマリインデックスは、開発時の利便性のためのものであると捉えることができます。実運用においては、プライマリインデックス定義を用いないことがベストプラクティスとなります。

■コラム:

アドホックなクエリのために、（本書の別の箇所で紹介している）アダプティブ（Adaptive）インデックスの利用を検討することができます。アダプティブインデックスの利用により、プライマリインデックスの特徴である全件スキャンと、それに伴う検索性能劣化、を避けることができます。ただし、アダプティブインデックスによってインデックス定義の自由度は増す事になりますが、それでも検索対象フィールドを特定する必要はあり、インデックス作成時に考慮されていなかったフィールドを用いて検索することもできる、プライマリインデックスのとは見なすことができる訳ではありません。

セカンダリインデックス

JSON ドキュメントの特定の要素（フィールドまたはドキュメントのキー）に対して設定するインデックスは、（バケット単位で定義されるインデック

第6章 クエリとインデックス

6.1 インデックス

スであるプライマリインデックスと区別して) セカンダリインデックスと呼ばれています。

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す。

トップレベルフィールド

```
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" }
```

述語条件の左項は、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。

ネストされたオブジェクト

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) を指定することができます。

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

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  
}
```

複合（コンポジット）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);
```

ドキュメントキーのインデックス

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);
```

インデックスのメタデータ

`system:indexes` に対してクエリすることにより、インデックスのメタデータを確認することができます。

利用例

```
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"
    }
  }
]
```


メタデータは、インデックスが存在するノード (`datastore_id`)、状態 (`state`) など、インデックスに関する情報を提供します。

■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

コミュニティエディションユーザーは、右のサイトを使って、インデックスアドバイザー機能を利用することができます：<https://indexadvisor.couchbase.com/indexadvisor> コミュニティエディションでは、Web コンソールの Query Workbench 画面に、「External Query Advisor」という上記サイトへのリンクが表示されます。

参考情報

Couchbase 公式ドキュメント Using Indexes^{*1}

^{*1} <https://docs.couchbase.com/server/current/learn/services-and-indexes/indexes/global-secondary-indexes.html>

6.2 インデックス設計

ここで解説する内容を理解することにより、Couchbase Server のクエリのパフォーマンスを向上させることができます。

カバリングインデックス

クエリがインデックス内のキーのみを参照している場合、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のみを利用してクエリに応答することができます。

検索に用いる必要のないキーをあえてインデックスに含めて定義することで、特定のクエリのパフォーマンスを向上させることができる一方、インデックスの維持に必要なリソース負担が増します。性能とリソースのバランスを考慮して最適化することになります。

パーシャル（部分）インデックス

CREATE INDEX DDL の定義に 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。

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

上記例では、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。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。

インデックスの WHERE 句では、より複雑な述語を使用できます。

パーシャル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特化したインデックス化範囲の限定による（更新・検索）性能の最適化
- 検索の選択肢が限定的な場合（性別や地域名など）に、複数のインデックスとして定義することによる性能の最適化（この場合、インデックス定義時のノード配置の指定と組み合わせて、分散環境の効果を活用することが考えられます）

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。ただし、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、大文字と小文字が混在している可能性のある `name` フィールドを検索するケースを想定しています。大文字と小文字の組み合わせの全ての可能性を検索するために、`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ためには、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 (`LOWER(name)`)、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";
```

この例では、条件式の右辺に、値が小文字で与えられています（念のために言えば、インデックスとの一致のために、条件式の右辺に LOWER 関数を使用することは必要ありません）。

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));
```

重複 (Duplicate) インデックス

異なる名前で、名前以外は、共通した定義を持つインデックスを重複して
定義することは許されています。

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"]};  
  
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。唯一の違いは、これらのインデックスの名前です。CREAT
E INDEX ステートメントの WITH 句を使用して、物理的な場所を指定しま
す。（指定しない場合、ランダムに決定されます）

クエリの実行時、最適化の一貫として、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。

エンタープライズエディションでは、下記のように複数のノード指定を行うことができます。

```
CREATE INDEX iDuplicate ON 'travel-sample'.inventory.airport(LOWER(name), id, icao)
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 句には、`node` の他に、`num_replica` (インデックスレプリカの数)、そして真偽値をとる `defer_build` (インデックスの遅延作成) があります。

配列インデックス

JSON は、スカラー、オブジェクト、または配列をフィールドとして持つことができます。各オブジェクトは、他のオブジェクトや配列をネストできます。各配列は他のオブジェクトや配列を持つことができます。

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。

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"  
  }  
]
```

`schedule` 配列に見られるような複雑な構造に対して、サブオブジェクト内の特定の配列または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_sched
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なっています。

```
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ように暗黙的に宣言された変数です。`v.day` は、配列の各要素であるオブジェクトの `day` フィールドを示しています。

これは一般化された式であるため、インデックスを作成する前にデータに追加のロジックと処理を適用する柔軟性を提供します。たとえば、各配列の要素に関連型インデックスを適用できます。

`DISTINCT` 修飾子により、インデックスからは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;
```

`ARRAY <インデックスキー> FOR v IN <配列名> END` 構文と合わせて利用することのできるキーワードを使って、複雑な条件指定を行うことができます。

配列から特定の条件で選択された要素の全てのフィールドを使ってインデックスを作成する例

週の最初の 4 日間にスケジュールされたサンフランシスコからのフライトのインデックスを作成します。

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

週初日にスケジュールされた「UA」フライトのリストを検索します。

```
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
```

上のインデックスは、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 クエリ述語中の `sourceairport = "SFO"` の部分は、インデックスの `WHERE` 句に一致します。

② クエリ述語中 `ANY-SATISFIES` 句の `'schedule'` 配列に関する検索のキーとして、インデックスキー `v.flight` が一致します。

③ クエリ述語中 `ANY-SATISFIES` 句のもう一つの条件 `v.day=1` は、インデックス定義の `WHEN` 句の条件「`v.day < 4`」に包含されます。

配列の全ての要素から選択されたフィールドを使ってインデックスを作成する例

フライトスケジュールのフライト番号とフライト日のみを使ってインデックスを作成します。

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

2 日にスケジュールされた「US681」フライトのリストを検索します。

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

簡略化された構文を使用して配列内のすべての要素からインデックスを作成する例

スケジュールのインデックスを作成します。

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

特定のスケジュールに一致するすべての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;
```

スケジュール配列の要素はオブジェクトであるため、述語条件の右側の値は同様にオブジェクトである必要があります。

N1QL がもつ、配列操作のためのキーワード `UNNEST` を使って、上のクエリを下記のように書き換えることもできます。

```
SELECT * FROM 'travel-sample'.inventory.route t
UNNEST schedule sch
WHERE sch = {"day":2, "flight": "US681", "utc": "19:20:00"};
```

アダプティブ (Adaptive) インデックス

`DISTINCT PAIRS` 構文を利用して、ドキュメントの指定された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。指定されたフィールドの順序にかかわらず、全ての組み合わせを用いてクエリを実行することが可能になります。

これにより、フィールドのさまざまな組み合わせに対してさまざまな複合インデックスを作成しなくても、効率的なアドホッククエリが可能になります。

これは、特殊なタイプの配列インデックスです。

```
CREATE INDEX 'ai_airport_day_faa'
```

```
ON 'travel-sample'.inventory.airport(DISTINCT PAIRS({airportname, city, faa}));
```

さらに 'self' キーワードを用いることにより、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));
```

性能最適化における考慮点

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、實際上、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）ことと裏返しの関係になります。

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 (バケット名) マッチ
- ② 条件句マッチ
- ③ 先行属性 (leading attributes) 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"
```

複数ドキュメント取得時の最適化

クエリにより、多数の (かつ大きな) ドキュメントが返される場合は、クエリで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 K-V API を使用してドキュメントをフェッチすることができます。すべてのデータ転送を実現するために、クエリプロセスのメモリ領域がボトルネックとなる事はありません。また、1 つの大きなネットワーク伝送ではなく、多数の小さ

なネットワーク伝送を用いることが可能になります（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ます）。

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

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

ここでは、META().id を検索の目的でなく、クエリ性能最適化のために利用しているため、カバリングインデックスの例とすることができます。このクエリでは、META().id を持たないドキュメントは存在しませんが、META().id IS NOT MISSING という条件は、インデックスとの一致（先行属性マッチ）のために必要になります。

6.3 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。

■コラム:

SQL 標準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。

基本

エスケープ文字

ユーザー定義語句（バケット名、フィールド名）は、「`」（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。Couchbase Server では、「-」は、予約語に含まれるため、「-」を含むバケット名を利用する場合、下記のようにバケット名をエスケープする必要があります。

```
SELECT ... FROM `travel-sample`
```

構造

N1QL は、JSON データを構成する文字列、数値、真偽値を（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。以下に具体的な例を見てみます。

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使うことができます。

{ } : []

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（インデックス、スライス）: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。

ドキュメントキーの利用

USE KEYS

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

```
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行う場合、インデックスを作成する必要がありません。

■コラム:

Couchbase Server では、ドキュメントキーが判明している場合には、ドキュメントキーを指定してドキュメントを取り出すことが、（RDB における、主キーでの検索のように）最も効果的であることは間違いあり

ません。一方でドキュメントキーが判明しているケースにおいては、性能を追求する場合、必ずしもクエリ用いて (Query サービスを介して) ドキュメントを取り出す必要はなく、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 (特に上記の例のような単純な利用の場合。一方、サブクエリを伴うような複雑なクエリの場合など、USE KEYS 句を効果的に利用できる場面があります)。

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 を用い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、後続する鉤括弧の中のドキュメントキーは完全一致が想定されていますが、META().id を利用する場合、N1QL クエリの一部として、LIKE と組み合わせることができ部分一致検索を行うことができる他、比較演算子による大小比較など、クエリの名で用いる際により自由度が高い表現と言えます。

また、検索条件としてのみではなく、SELECT META().id FROM ... WHERE ... 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。

上記のクエリ中の `META().id` の部分は、下記のようなキースペース (の別名) を含んだ表現 (`META(a).id`) を用いることができます。このクエリのように、クエリの中でキースペースが一意に決まっている場合、指定は必須ではありません。複数のキースペースが混在している中で、ドキュメントのキーを用いる (取り出す) 場合には、キースペースを指定した表現を用います。

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

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 で検索された結果は、JSON データとして、フィールド名と値のペアとなることが基本ですが、`RAW` キーワードを `SELECT` 句と共に用いることで、フィールドの値のみを取り出すことができます。

RAW を用いない例

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

結果は、JSON データ (フィールド名と値のペア) の配列になります。

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

RAW を用いた例

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

結果は、値の配列になります。

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

```
"Aberdeen",  
"Abilene"  
]
```

DISTINCT と組み合わせた利用例

```
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"  
]
```

配列からデータを取り出す

ARRAY または FIRST から始まり、END で基本単位となります。IN または WITHIN と FOR により、コレクションの要素へのループを表現しています。

この構文を利用して、ドキュメントに含まれる配列が要素として、オブジェクト（サブドキュメント）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（配列の要素に対して `map` 操作を実行します）。

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います。

WHEN

WHEN 句により、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
```

下記の通り、上記説明した要素の組み合わせにより複雑なステートメント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
```

配列を検索条件に用いる

ANY または EVERY から始まり、END までで基本単位となります。SATISFIES で条件を指定します。

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

■コラム:

この構文は、配列から、真となった要素を取り出しているのではないことに注意してください。これは、SQL の WHERE 条件が検索に使ったカラムを取り出すものではないのと同様です。

ANY

配列に条件で指定した要素が一つでも含まれる場合、真となります。

```
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。

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

IN または WITHIN

IN 句が、指定された配列のトップレベルの要素を検索するのに対して、WITHIN 句は、現在の配列とその子、および子孫を含めて検索対象とします。

ドキュメントキーによる結合: NEST

NEST は、外部の子ドキュメントを親の下に埋め込む特別なタイプの JOIN です。ドキュメント・キーを介した参照関係でデータをモデル化した際に用いる事ができます。ON KEYS により、対象とするドキュメントのキーを指定することができます。

以下の 2 つのコレクションとドキュメントに対するクエリについて考えてみます (データの構造および内容は、あくまでこの箇所での説明のためのものであり、他の箇所と同じでないことにご注意ください)。

コレクション: retail.east.order

```
{
  "order_id":1234,
  "customer_id":"34567",
  "total_price":"65.5",
  "lineitems":["o11","o12","o13"]
}
```

コレクション: retail.east.lineItem

```
{
  "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"
}
```

```
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、下記のように、一つのドキュメントになります。

```
[  
{  
  "items": [  
    {"item_id": "789", "qty": "3"},  
    {"item_id": "234", "qty": "5"}  
  ],  
  "order_id": "1234"  
},  
{  
  "items": [  
    {"item_id": "899", "qty": "8"},  
    {"item_id": "651", "qty": "2"}  
  ],  
  "order_id": "9812"  
}  
]
```

サブドキュメントのフラット化: UNNEST

UNNEST は、ネストされたオブジェクトを最上位ドキュメントとして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（受注明細データを中に含む受注伝票データ）。


```
{
  "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

クエリでは、UNNEST をサブドキュメントに対して指定し、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
```

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 }
]
```

(JSON 中の要素の出現順はこの通りではありません)

■コラム:

JSON オブジェクトのフィールドは並び順に意味を持ちません。

データ型: NULL, MISSING

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、馴染みがあるところです。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？「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

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。データベースのコンテキストでいえば、テーブルスキーマにおけるカラムは、一定のデータ型を持ちますが、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」の場合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 ドキュメントでフィールド (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 における SELECT 文で、検索結果として返されるフィールドが存在しない (MISSING である) データがある場合、(クエリ処理において、内部的には MISSING はリテラル式で扱われますが)、最終的な結果の配列中では、(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。

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。

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。

Couchbase 公式ドキュメント Data Types MISSING^{*2}

比較演算: IS [NOT] NULL|MISSING

IS [NOT] NULL|MISSING ファミリー演算子を使用すると、データ・セット内の属性の存在（または不在）に基づいて条件を指定できます。

これらは、データ型の定義から導か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/datatypes.html#datatypes>

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

IS MISSING

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

```
{  
  "results": [  
    {  
      "fname": "Harry"  
    },  
    {  
      "fname": "Jane"  
    }  
  ]  
}
```

比較演算: IS [NOT] VALUED

NULL と MISSING は、JSON データ上明らかに異なる状態であり、区別される必要がある一方、アプリケーションにおける検索条件としては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、TRUE (真) になります。
- IS NOT VALUED は、値が'MISSING' あるいは'NULL' である場合、TRUE (真) になります。

参考情報

Couchbase 公式ドキュメント Comparison Operators^{*3}

SQL と比べた場合の制約

- 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 連結の最初の (または唯一の) クエリである必要があります

■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)
- コストベースオブティマイザ (CBO)
- 無制限のクエリ同時実行

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

- 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

6.4 結合

Couchbase Server の N1QL で利用することのできる、以下の 3 つの結合のタイプについて解説します。* ANSI JOIN * Lookup JOIN * Index JOIN

下の二つの JOIN では、混同しやすい Couchbase Server 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。

ここでは、これらの用法について、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。

データモデル仕様

- 「ルート」と「エアライン (航空会社)」という、二つの種類のデータモデルがある
- 「ルート」モデルのデータ構造は、「エアライン」への参照（複数）を含む。参照は、「エアライン」ドキュメントのキーと一致している。

ANSI JOIN

ANSI JOIN でドキュメントキーを用いた結合を行うためには、META().id を利用し、適切なインデックスが作成されている必要があります。

シンタックス

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

クエリ例

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

Lookup JOIN

ON KEYS 句を用います。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。

シンタックス

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

クエリ例


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

Index JOIN

ON KEY ... FOR 句を用います。下記のクエリ例では、Lookup 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ください。Index 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。

シンタックス

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

クエリ例

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

参考情報

Couchbase 公式ドキュメント JOIN Clause^{*4}

6.5 レンジスキャン

分散データベース一般のレンジスキャン

分散データベースにおいては、一般的に（キーを用いた）レンジスキャンを効果的に行うためには、論理的に関連性のあるキーを設計するのみではなく、あらかじめそのレンジに含まれるデータが、同じ、あるいは近い場所（ノード、リージョン、パーティション）に存在する様に、（物理）設計を行う必要があります。

例えば、MongoDB では、シャーディングの方法として、ハッシュとレンジのいずれかを要件に応じ選択することになります（この選択が物理配置に影響します）。HBase では、データは RowKey で（物理的に）ソートされます。

Couchbase Server の特殊性

Couchbase Server は、N1QL という SQL を JSON のために拡張したクエリ言語で、検索を行うことができます。その際、重要なことは、Couchbase

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

Server の中心的なアーキテクチャーは KVS(キーバリューストア) であり、そのレベルでは、バリューとして JSON データを格納しているのに過ぎないため、検索を行うための前提として、インデックスの定義・構築が行われている必要があります。

そもそも、スキーマレスなデータストアに対して、どうやって検索が可能なのか (全件サーチするのではなければ)? という観点からは、これはむしろ納得できるものとして受け止められるかと思います。

ただし、ここで留意が必要なのは、KVS のキーを使った検索 (つまりキーを特定したアクセスではなく、ある範囲のキーを指定してデータを一度に取得するという要件) であっても、事前にインデックスの作成が欠かせない、ということなのです。このことをよりよく理解するために、以下の二つの背景を押さえておくことが重要です。

- Couchbase Server においては、クエリおよびインデックスサービスは、(KVS である) データサービスとは、アーキテクチャー上完全に独立している (利用する要件がない場合は、無効にすることができ、そのために潜在的にリソースが消費されることはない)。
- Data サービスのキーのシャードイング方法には、ハッシュが用いられており、ユーザによって変更する余地を残していない。(CouchbaseServer は、メモリファーストアーキテクチャーと、クラスターマップを用いたクライアントからのノードへの直接のアクセスによって、データをチャンクで取り出すよりも、高い性能を実現している)

N1QL によるレンジスキャン

インデックスの作成

```
CREATE INDEX range_index ON MyFirstBucket(META().id);
```

ここで、`META().id` が、KVS のキー（ドキュメント ID）に対応します。

`MyFirstBucket` は、インデックスを作りたいバケットです。

インデックス作成のための構文は、`CREATE INDEX` [インデックス名] `ON` [キースペース名]([ドキュメントのフィールド名]) というものです。バケット名の後のカッコ内に通常 JSON ドキュメントのフィールド名を指定しますが、ここではドキュメントの内容ではなく、ドキュメントが格納される際のキー/ID をインデックスとして利用するため `META().id` という特殊なキーワードを用いています。

ここではキースペースの指定に `MyFirstBucket` というバケット名のみを指定していますが、この場合、そのバケットの `_default` スコープの `_default` コレクションを指定したことになります。

利用例

下記のような、キーにレンジを指定した検索が可能になります。

```
SELECT * from MyFirstBucket where Meta().id BETWEEN 'ABC:001' AND 'ABC:999';
```

また、下記のような、キーの一部を指定した曖昧検索も可能になります。

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

Data サービス API によるレンジスキャン

Couchbase Server で「キー」のみによるレンジスキャンを行うケースとして、下記の様な Data サービスへの API 呼び出しとして実現することが考えられます。(ここでは、Java を用いたコード例を示しています)

同期型の例

```
protected List<Object> get() {  
    for (String key : keys) {  
        GetResult result = collection.get(String.valueOf(key));  
        listResults.add(result);  
    }  
    return listResults;  
}
```

非同期の例

```
protected List<Object>get() {  
    ReactiveCollection reactiveCollection = collection.reactive();  
  
    Flux<Object> resultFlux = Flux.fromArray(keys.toArray())  
        .flatMap( k -> reactiveCollection.get(String.valueOf(k)));  
  
    List<Object> listResults = resultFlux.collectList().block();  
    return listResults;  
}
```

```
}
```

上記のコードは、コード外で、‘id‘ のリストを作成し、‘keys‘リストとして与えられることを想定しています。

6.6 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。

ユーザー定義関数には次の 2 つの種類があります。

- **インライン関数**は、サブクエリを含む N1QL 式を使用して定義されます。クエリを簡素化するために、複雑な式や、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) サポートされている言語は JavaScript のみです。

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** 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。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)。

■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。

参考情報

Couchbase 公式ドキュメント User-Defined Functions^{*5}

Couchbase 公式ドキュメント CREATE FUNCTION^{*6}

^{*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>

Couchbase 公式ドキュメント EXECUTE FUNCTION^{*7}

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

第 7 章

アーキテクチャー

7.1 分散システムアーキテクチャー

分散アーキテクチャーの特徴

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・可用性に優れています。

分散アーキテクチャーは、複数のサーバを用いて、一つのプラットフォームを構成します。これを一般に**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は**ノード**と呼ばれます。Couchbase Server は、そのトポロジー（ノード構成）に関わらず、クラスターとして柔軟に構成可能です。

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、名前ノード、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。その意味で、すべ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デプロイされますが、開発などの目的で、すべての機能を単一のノードで実行することもでき、その場合にもアーキテクチャーや挙動については変わるところがありません（スタンドアローンモー

ドのような通常とは異なる特別のデプロイ形態は存在しません)。

クラスターの構成変更

クラスターは、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。

ノードが追加される際に、ユーザーは、そのノードで実行されるサービスを構成します。ユーザーは、あるノードがクラスターの一部としてはじめに構成される（クラスターに参加する）際に、そのノードで特定のサービスを有効または無効にできます。

■コラム:

既にクラスターを構成しているノードのサービスを変更することはできません。そのようなことを実現したい場合には、一度そのノードをクラスターのメンバーから除いた後に、改めて参加させる際にサービスを選択します。

ノードは、簡単に追加または削除できます。Data サービスノードの追加・削除は、すべてのノードにデータを均等に再分散するリバランスプロセスを通じて、実現されます。リバランスプロセスはオンラインで行われ、アプリケーションのダウンタイムは必要ありません。Query サービスは、ステートレスであり、クラスター構成変更の影響を受けません。Index サービスが管理するインデックスのノードへの配置は、ユーザーによりカスタマイズすることが可能であるため、クラスター構成変更時の影響関係がユーザーによって把握されている必要があります。

ワークロード分散

ノードは、特定のワークロード〜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〜の処理専用に構成することによってワークロードを分離し、クラスターを各サービスの要件・特性に応じたりソース配分にて構成することができます。このリソース配分には、サービス当たりのノード数のみではなく、各ノードでワークロード/サービスの特性に応じたりソース (メモリ、CPU) を割り当てることも含まれ、こうした水平および垂直に構成可能なスケーリングは、**マルチ・ディメンショナル・スケーリング (MDS)** と呼び習わされています。

■コラム:

MDS は、エンタープライズエディションのみサポートされるとされています。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せん。つまり、エンタープライズエディションでは、Query サービス等をそのサービス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。これに付随して、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、Data サービスとの共存が前提であることから、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約はない、と言った違いが見られます。

レプリケーション

CouchbaseServer はアクティブなデータのコピー（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、すべてのコピーが別々のノードに配置されるようにします（ピアツーピアレプリケーション）。Couchbase Server は、最大 3 つのレプリカ（つまり、最大 4 つのデータコピー）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（複数の）ノードに複製するため、単一障害点を排除します。

ノードがダウンした場合、Couchbase Server は、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、単一ノード上のデータの損失から保護し、ハードウェア障害やサービスの中断からの回復を可能にすることでデータの可用性を向上させるのに役立ちます。

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。これにより、分散環境に起因する、データの一貫性を維持するためのオーバーヘッドが取り除かれ、非常に高いスループットと低遅延が可能になります。1 つのノードに対する 1 秒あたり数十万の要求の 99 パーセンタイルで、1 ミリ秒未満の応答時間を測定することは珍しくありません。

一方、レプリカへのアクセスのための API も提供されており、アクティブコピーが利用できなくなってからフェイルオーバーが実行されるまでの期間、アプリケーションからその API を用いて、読み取りの可用性を向上させることができます。

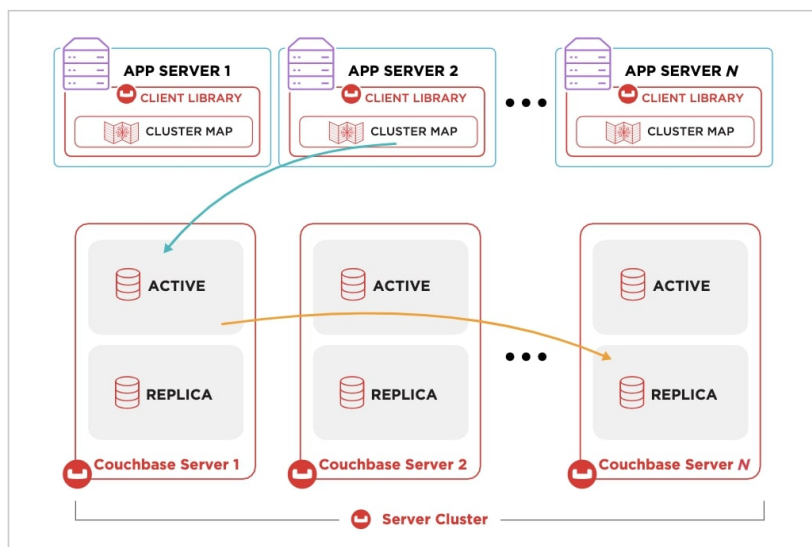


図 7.1: replica concept

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

データの分散管理 (シャード)

Couchbase Server は、内部的に、**vBucket** (シャードまたはパーティションと同義) を使用して、データを管理しています (ユーザー/アプリケーションは vBucket を直接操作しません)。一つのバケットは、1024 個のアクティブ vBucket と、(1 レプリカに対して) 1024 個のレプリカ vBucket に分割して管理されています。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

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

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。

スマートクライアント

Couchbase Server のクライアントは、接続されている Couchbase Server クラスターのトポロジを自動的に認識し、トポロジに変更が発生しても、変更を透過的に維持します。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（ブートストラップ）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の接続中、クラスターマップは最新の状態に維持されます。このような機構は、**スマートクライアント**と呼ばれています。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。

Data サービスと対話する際、クライアントは、何らかのプロキシやルーティングエンティティに問い合わせることなく、任意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。これは、Couchbase Server からのトポロジ変更通知に基づいて、クラスターマップのローカルコピーを保存し、透過的に更新することによって、機能します。

Data サービスにおいて、データ（ドキュメント）は、格納されているバケットにおいて一意のキー（ドキュメントキー、またはドキュメント ID）を持ちます。キーは、ユーザー/アプリケーションにより自由につけることができますが、内部的には、CRC32 ハッシュアルゴリズムを用いて変換されています。このアルゴリズムにより、バケットを構成する vBucket 中のデータ（ドキュメント）の分布は、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ために、ドキュメントのキーに対して、CRC32 ハッシュアルゴリズムを使用して、どの vBucket がそのキーを担当しているかを識別します。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。

同様のトポロジ認識を使用して、Query サービスを実行しているノードのリストが維持されます。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。

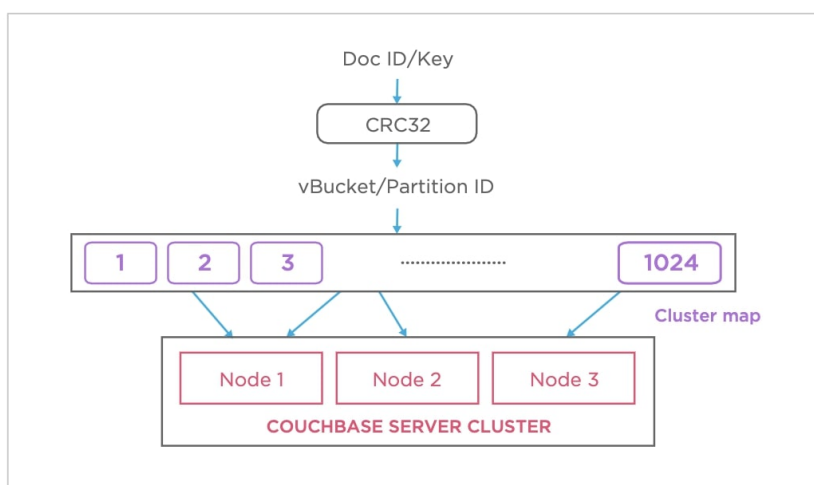


図 7.2: cluster map

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

データ転送

Couchbase Server は、その内部で固有のストリーミングプロトコル、データベース・チェンジ・プロトコル (DCP) を用います。DCP は、クラスター内部、および、クラスター外部に対して、Couchbase Server で生じたデー

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

第 7 章 アーキテクチャー 7.2 XDCR(Xross Data Center Replication)

タの変更を伝える役割を持ちます。

Couchbase Server 内では、DCP を介して、レプリカ、インデックスなどを更新します。また、DCP は、Spark、Kafka、Elasticsearch などの外部システムとの統合のためのコネクタへデータをフィードするためにも用いられます。また、他の Couchbase Server クラスターとのデータ同期の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、あるデータの変更がメモリー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れた時点から再開します。

7.2 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 (バージョン 7.0 より。バージョン 6.6 までは、一部機能を除き利用できます)。

XDCR は、何故必要か？

XDCR(クロスデータセンターレプリケーション) 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、ハイブリッド、コンテナなどのプラットフォームに関係なく、任意の Couchbase Server クラスター間でデータを複製できます。

第7章 アーキテクチャー 7.2 XDCR(Xross Data Center Replication)

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。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、クラスター内部の構成変更に対して透過的に動作します。計画的ないし障害によるノードの追加・削除（トポロジ変更）中であっても、マニュアルの対応は不要であり、継続して動作し続けます。

Couchbase Server は、「データベース変更プロトコル（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 DCP のコンシューマーであり、DCP に依存しています。

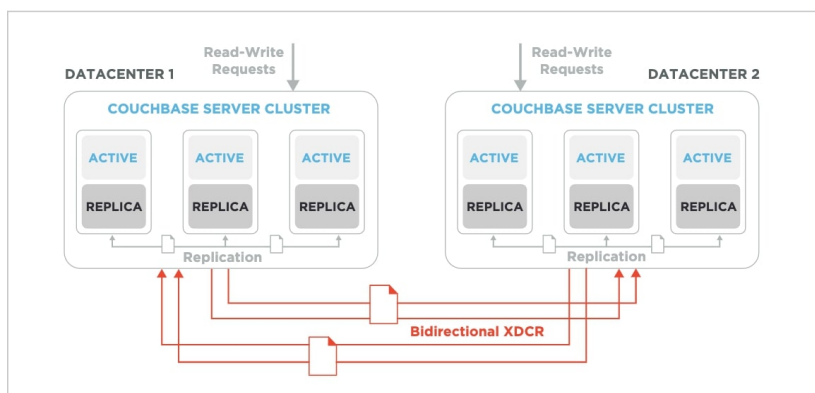


図 7.3: XDCR

(画像は、Couchbase Under the Hood: An Architectural Overview^{*3}よ

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

第 7 章 アーキテクチャー 7.2 XDCR(Xross Data Center Replication)

り引用)

機能ハイライト

チェックポイントによるレプリケーション中断・再実行制御

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。レプリケーションの中断後、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。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が考えられます。

チェックポイント更新の間隔が短いほど、チェックポイントの精度が高くなり、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はシステムリソースを使用し、パフォーマンスに影響を与える可能性があります。

複数のレプリケーション間の優先度設定

(あるバケットについて) レプリケーションが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、既存レプリケーションでは差分データ更新が行われているのに対して、新規レプリケーションでは全件データのレプリケーションが一斉に実行されることにより)。XDCR では、レプリケーションに対して優先度を設定することができ、このようなケースにおいても、優先度に基づいて適切なリソース配分を実現することが可能です。

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や、ユーザーにより設定 (例えば、高度な設定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、ネットワークの帯域幅を節約するために高度に最適化

第 7 章 アーキテクチャー 7.2 XDCR(Xross Data Center Replication)

されています。

レプリケーション対象データのフィルタリング

Couchbase Server のエンタープライズエディションでは、レプリケーションの対象とするデータを、ユーザが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: エディション間の差異

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。

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、特に既存のサービス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が考えられます。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も考えられます。

参考情報

Couchbase 公式ドキュメント XDCR Advanced Settings^{*4}

^{*4} <https://docs.couchbase.com/server/current/xdcr-reference/xdcr-advanced-settings.html>

第 7 章 アーキテクチャー 7.2 XDCR(Xross Data Center Replication)

Couchbase ブログ Understanding Cross Datacenter Replication (XDCR) -- Part 1^{*5}

^{*5} <https://blog.couchbase.com/understanding-xdcr-part-1/>

第 8 章

クラスター

8.1 フェイルオーバー

分散アーキテクチャーのフェイルオーバー

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一般におけるフェイルオーバーについて、一般に妥当すると考えられる前提や要件などを整理します。

まず、広い意味でのフェールオーバーとして、以下の 2 種類があります。

計画的なノードの削除（または、グレースフルフェイルオーバー）

クラスタから機能しているノードを削除するには、管理コンソール等を利用し、削除するノードを指定します。その後クラスタをリバランスし、そのノードへのデータ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。完了時点で、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ようなことも、データを消失してしまうこともなく、ノードを削除してクラスタを（グレースフルに）リバランスできます。

機能しているノードを運用管理目的で削除したい場合、フェイルオーバーではなく、ノード削除 (Remove) の後、リバランスを実行します。

障害時のノードの削除（または、ハードフェイルオーバー）

フェイルオーバーとは、そのノードをクラスタから削除し、他のノードにあるレプリケートしたデータを利用可能にすることです。

フェイルオーバーは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。（機能しているノードをフェイルオーバーした場合、フェイルオーバーは即座にノードをクラスタから削除するため、データロスが発生する可能性があります。）

ノード障害時に他のノードへレプリケート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（フェールオーバーしたノード上で、ディスクに永続化されていた場合、そのサーバーが復旧出来た場合には、データを復旧できる可能性があります（デルタリカバリー）。そうでない場合データは、完全に消失することになります。

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。

フェイルオーバー手段

ノード障害発生時には、フェイルオーバーを手動で実行するか、あるいは、自動的にフェイルオーバーをトリガーし、縮退モードでクラスタを稼働させるように設定することが考えられます。フェイルオーバーはクラスタの性能を縮小させてしまうため、フェイルオーバーの状況をどのように処理すべきかをよく検討する必要があります。

自動フェイルオーバーを利用すると、ユーザ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を管理する場合、クラスタを監視し、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（あるいは外部スクリプト）でフェイルオーバーを実行します。この手順では、よ

り多くの監視や手動の操作が必要です。

復旧

フェイルオーバー実行により、クラスタが縮退運転している間、クラスタ内に残る稼働中のノードに対する負荷は増大します。クラスタがノード障害前と同じ状態で機能するためには、正常に機能するノードをクラスタに戻し、リバランスを実行して、ノード障害から復旧する必要があります。

手動でフェイルオーバーを実行しても、クラスタが自動的に実行するようにしても、障害の原因を解明する必要があります。そして、正常に機能するノードをセットアップし、そのノードを追加し、クラスタをリバランスします。

フェイルオーバーシナリオに対処する際の、ノードの交換および追加に関する選択肢を以下に記します。

- ハードウェアやシステム障害が起因してノードがダウンした場合、新規の交換用ノードをクラスタに追加してリバランスする。
- クラスタのキャパシティ不足が原因でノードがダウンした場合、ノードの交換に加え、必要なキャパシティとするために必要な分だけノードを追加する。
- ノードの障害が一時的なものである場合、そのノードをクラスタに再追加します。

連鎖反応 (Thundering Herd)

自動でコンポーネントを取り除くには、どんな分散システムでも問題が付き物です。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問題を解決するどころか、より多くの問題を発生させる可能性があります。次のような状況では、問題を誘発する可能性があります。

クラスタが、キャパシティの 80-90% で稼働しているとします。その時

点で、順調に稼働していたとしても、クラスターのキャパシティは限界です。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 4 ノードでは負荷が増大し、正常に処理することはできないでしょう。結果として負荷が増加することで、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ます。これらの障害は連鎖し、結果的にクラスター全体の消失へとつながる可能性があります。

この場合の次善策は、単一ノード障害が発生してもクラスターの運用を続け、新しいサーバをクラスターに追加し、失われたキャパシティを補填して、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。こうすれば、クラスター全体が利用不可となる代わりに、部分的な障害で済みます。単一ノード障害発生時には、一部のリクエストを処理できない方が、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いことよりも良いと言えます。

このような状況の予防策としては、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することです。

実行可能条件

ネットワークデバイスの故障が原因でネットワークが分割されてしまう、ネットワーク分断やスプリットブレインの状況を考慮し、多くのクラスター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。これは、ネットワーク分断が発生した際に、2 ノードのクラスターがお互いのノードをフェイルオーバーすることを防ぎます。クラスターによって、データのセットが構成されている場合、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レプリカデータにより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発動す

る。これは、フェイルオーバーの連鎖による、以後の性能や安定性の劣化を防ぐためです。多くの場合、クラスタが機能できなくなるまで劣化し続けるよりも、データセットの小さな部分にアクセスできなくなる方が望ましいでしょう。

- 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。

フェールオーバーの運用

手動フェイルオーバーを行う場合、二つ方法があり、一つは人による監視、もう一つは外部システムを利用する方法です。

- **保守要員による運用** アラートに対して次の行動に関する意思決定をする人員を確保することは一つの選択肢です。

人間が広範囲のデータ、観測、経験を考慮し、最適な方法で状況を解決できるユニークな存在です。多くの組織では、人が影響に対する考慮をしないままにフェイルオーバーを自動化することを許可していません。マニュアルによる介入を行うことの問題点はコンピュータベースのモニタリングシステムを利用することに比べ、対応が遅くなるということです。

- **外部モニタリング** もう一つの選択肢は REST API を利用してクラスタをモニタリングするシステムを利用することです。

このような外部システムは、クラスタ以外のシステムコンポーネントに関しても考慮できるため、ノードのフェイルオーバーにとって優位な位置に存在します。

例えば、モニタリングソフトウェアは、クラスタ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できます。システムは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、ノードのフェイルオー

バーはしません。

モニタリングシステムはクラスター周辺のコンポーネントが機能しているか、クラスター内の様々なノードが健全であるかを判断することもできます。モニタリングシステムが、問題は単一のノードだけで起きており、クラスター内の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すれば、システムから REST API やコマンドラインツールを利用してそのノードをフェイルオーバーすることもできます。

Couchbase Server フェイルオーバー機能

自動フェイルオーバー仕様・制約

Couchbase Server の自動フェイルオーバーには多くの制約があります。これは自動フェイルオーバーを利用する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トリガーされません。

- **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** 管理者が指定したイベントの最大数までしか、自動フェイルオーバーはトリガーされません。許可される最大の最大値は 3 です。この自動フェイルオーバーの最大数に達すると、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発生しません。ただし、最大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発生しません。たとえば、バケットにレプリカがない場合はこのケースです。
- **非フェールオーバーノードの応答状況** (フェールオーバー対象となる) ノードが応答しなくなった後でも、ノードの過半数に接続できる

場合のみ、自動フェイルオーバーがトリガーされます（グループの場合、グループが応答しなくなった後でも、グループの過半数に連絡できる場合のみ）。

デフォルト設定

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。

待機時間

ノードがフェイルオーバーされるまでに、最低でも 30 秒の遅延時間が必要です。この時間を延ばすことはできますが、ソフトウェアはダウンしているかもしれないノードに対して複数回の死活監視を実行するようにハードコーディングされています。これは機能しているが遅いノードのフェイルオーバーや、ネットワーク接続問題によりフェイルオーバーが発動することを防ぐためです。

通知

REST API を利用して、ノード障害が発生し、ノードが自動でフェイルオーバーされた際に Couchbase Server が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。

サービス固有の自動フェイルオーバーポリシー

応答しないノード上の 1 つまたは複数のサービスのサービス固有の自動フェイルオーバーポリシーに準拠して自動フェイルオーバーがトリガーされます。

■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、サービス毎にノードへ配置を構成するマルチディメンショナルスケーリング (MDS) が可能です。MDS を利用している場合、ノードの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。コミュニティエディションでは、全てのサービスが全てのノードで稼働することになるため、Data サービスの仕様が満たされている必要があります。

Data サービス

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

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

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。

グループフェイルオーバー

ノードをラック (アベイラビリティゾーン) 毎に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。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

クラスター単位で有効化することにより、ディスク障害に対するフェイルオーバーの機能を活用することができます。デフォルトでは無効とされています。

■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。

フェイルオーバー実行方法

Couchbase Server のフェイルオーバーは以下の方法で実行できます。

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、クラスターが Down として判定したノードのみ 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ます。
- 確認・警告メッセージが表示されます。

- FailOver をクリックし、ノードをフェイルオーバーします。Cancel を選択することもできます。

コマンドラインツール

couchbase-cli の failover コマンドを利用して、複数のノードをフェイルオーバーできます。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で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
```

成功すると、ノードがフェイルオーバーされたことを示すメッセージが表示されます。

8.2 コミュニケーション概要

Couchbase Server におけるネットワーク間通信の種類について整理します。

Couchbase Server は、クライアントからクラスター、ノードからノード、およびクラスターからクラスターへの通信を処理します。また、サードパーティ製品への接続も提供します。

クライアントとクラスター間

クライアントアプリケーションは、サーバー定義の（複数の）アクセスポイントを介して Couchbase Server クラスターと通信します。各アクセスポ

第 8 章 クラスター 8.3 クライアント・クラスター間コミュニケーション

イントは、非暗号化の通信と暗号化通信の両方にポートを提供します。

ノード間

クラスターノードは、データの複製、インデックスの維持、ノードの状態の確認、クラスター構成への変更の伝達などのために相互通信します。

クラスター間

Couchbase Server-クラスターは、クロスデータセンターレプリケーションを使用して相互に通信します。

コネクター

Couchbase Server クラスターはサードパーティ製品と通信します。コネクタ。Elasticsearch、Hadoop、Kafka、Spark、および Talend 用のコネクタが提供されています。ODBC および JDBC 用のドライバーが提供されています。

8.3 クライアント・クラスター間コミュニケーション

クライアントとクラスター間のコミュニケーションについて解説します。

クライアント接続における 3 つのフェーズ

クライアント接続は、認証と認可 (承認)、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。

第 8 章 クラスター 8.3 クライアント・クラスター間コミュニケーション

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。

検出 (Discovery)

クラスターマップがクライアントに返されます。これは、現在のクラスターポートロジを示しています。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配置、およびノード間のデータ配置に関する情報を含みます。

クライアントは、これらの情報を持っているため、適切なノードへアクセスすることができます。これは、単に適切なサービスを提供するノードへアクセスできることを意味しているだけでなく、データの作成や（検索クエリではなく、キー/ID による）取得の際に、そのデータの管理を担当するノードへ直接アクセスができることを意味しています。

ここまでの処理は、ブートストラップと呼ばれます。

サービスへの接続

クラスターマップを取得すると、クライアントはサービスレベルの操作を実行するために必要な接続を確立します。

この時実行しようとする操作の種類・内容によって、認可が必要になる場合があります。クライアントが要求しているリソースへのアクセスに対する適切な権限に関連付けられているロールを、ユーザーが持っている場合、アクセスが許可されます。

クラスターポートロジが変更された場合、サービスへの接続要求の際に例外が発生する可能性があります。この場合、検出を再実行し、新しい接続で操作を再試行する必要があります。

第 8 章 クラスター 8.3 クライアント・クラスター間コミュニケーション

他の分散アーキテクチャーとの違い

上記の中でも、特に「検出」のプロセスの部分は、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と比べて、Couchbase Server に独自の部分だといえます。

Couchbase Server とは異なり、多くの分散アーキテクチャーを持つデータベースでは、クライアントとのコミュニケーションを担当する特別なノードが存在しています。この違いは、(メモリファーストアーキテクチャに加えて) Couchbase Server が、低遅延・ハイスループットを実現する要素となっています。



第 9 章

運用

9.1 監視

Couchbase Server における監視手法について見ていきます。

はじめに、Couchbase Server の監視機能について説明します。その後、外部の監視システムとの連携の例として、Prometheus および Grafana との連携について紹介します。

Couchbase Server 監視機能

監視対象

Couchbase Server の監視機能の対象は、CPU・メモリ利用量などハードウェア情報に関するノード（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に関するクラスターやバケット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 コンソールには、様々な情報を期間やレベル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。

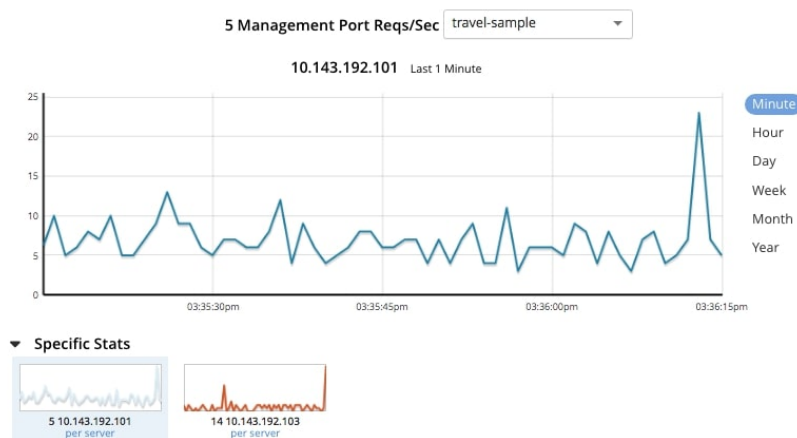


図 9.1: web console monitoring

Couchbase 公式ドキュメント Monitoring with UI^{*1}

E メールアラート

Couchbase Server は、(自動フェールオーバー等) 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。

外部監視システム連携

Rest API と CLI

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 と CLI 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は、全て Rest API を使って取得する

^{*1} <https://docs.couchbase.com/server/current/manage/monitor/ui-monitoring-statistics.html>

ことが可能です。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` があります。

Prometheus

Prometheus 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。そして、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。下記で、公開されている Exporter のリストを見ることができます。

Prometheus 公式ドキュメント Exporters and Integrations^{*2}

Couchbase Github リポジトリ couchbase-exporter^{*3}

Couchbase Exporter は、監視する Couchbase Server クラスターに対して、プロセスを実行します。

```
./couchbase-exporter --couchbase.username Administrator --couchbase.p
```

Grafana

Grafana のダッシュボード定義のサンプルが下記で公開されています。

Couchbaselabs GrafanaDashboard.json^{*4}

^{*2} <https://prometheus.io/docs/instrumenting/exporters/>

^{*3} <https://github.com/couchbase/couchbase-exporter>

^{*4} <https://github.com/couchbaselabs/blog-source-code/blob/master/Meliani/GrafanaDashboard.json>

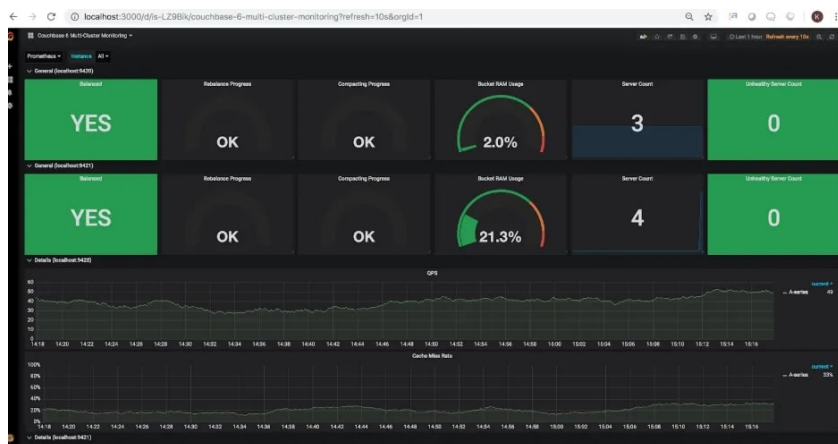


図 9.2: grafana

参考情報

Couchbase ブログ Couchbase Monitoring^{*5}

Couchbase ブログ Couchbase Monitoring Integration with Prometheus and Grafana^{*6}

9.2 バックアップ: cbbakupmgr

どんなデータベースにとっても、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、cbbakupmgr コマンドの使用例を紹介します。

このチュートリアルの実行例では、travel-sample パケットと beer-sa

^{*5} <https://blog.couchbase.com/monitoring-couchbase-cluster/>

^{*6} <https://blog.couchbase.com/couchbase-monitoring-integration-with-prometheus-and-grafana/> Couchbase と Prometheus, Grafana との連携に関する詳細な手順説明

ample バケットを利用します。

バックアップリポジトリの作成

cbbakupmgr 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、複数のバックアップリポジトリが含まれることになります。バックアップリポジトリを構成する最も簡単な方法は、単一のクラスターに直接対応させることです。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されます。このチュートリアルでは、/data/backup にあるバックアップアーカイブを使用します。指定されたディレクトリが空の場合、バックアップアーカイブは自動的に作成されます。以下は、「cluster」という名前でバックアップリポジトリを作成する例です。

```
$ cbbakupmgr config -a /data/backup -r cluster
```

```
Backup repository 'cluster' created successfully in archive '/data/backup'
```

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、バックアップ方法を変更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。これを行うためには以下のオプションを使い、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。

```
$ cbbakupmgr config -a /data/backup -r single \
```

```
--include-data travel-sample --disable-data
```

```
Backup repository 'single' created successfully in archive '/data/backup'
```

`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。使用可能なオプションとその使用方法の詳細については、ドキュメントをご参照ください。

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がどのように見えるかを確認します。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ディレクトリを表示します。ここでは、バックアップはまだ作成されていないので、このコマンドの出力でアーカイブ情報のみが確認できます。各リポジトリに含まれるディスク容量に関する情報もあります。

バックアップの作成

バックアップリポジトリが構成されたので、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すべての構成情報が含まれているため、バックアップ作成時には、バックアップリポジトリ名とバックアップするターゲットクラスターの情報を指定するだけです。以下は、「cluster」バックアップリポジトリでバックアップを作成する方法の例です。localhost でクラスターが実行されていると想定します。

```
$ cbbakupmgr backup -c 127.0.0.1 -u Administrator -p password -a /data/backup -r cluster
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、プログレスバーは完了までの推定時間を示しますが、バックアップが完了すると、これは平均バックアップレートに変更されます。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」というメッセージと、バケット

第9章 運用

9.2 バックアップ: cbbakupmgr

ごとにバックアップされた変更と削除の内訳が表示されます。

「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。

```
$ cbbakupmgr 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` バケットのインデックス定義をバックアップするようにのみ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が変更されたことを確認してみましょう。

```
$ cbbakupmgr info -a /data/backup --all  
Name | UUID | Size | # Repos |  
backup_repo | 32c97d5f-821a-4284-840b-9ee7cf8733a3 | 54.35MB | 2 |  
  
* Name | Size | # Backups |  
* cluster | 54.33MB | 1 |
```

```

+ Backup | Size | Type | Source | Cluster
+ 2020-03-25T08_08_11.770436Z | 54.33MB | FULL | http://localhost:9000 | ffa8024f

- Bucket | Size | Items | Mutations | Tombstones | Views | FTS | Ind
- beer-sample | 18.43MB | 7303 | 7303 | 0 | 1 | 0 | 1

- Bucket | Size | Items | Mutations | Tombstones | Views | FTS | Ind
- travel-sample | 35.91MB | 31591 | 31591 | 0 | 0 | 0 | 1

* Name | Size | # Backups |
* single | 16.20KB | 1 |

+ Backup | Size | Type | Source | Cluster
+ 2020-03-25T08_08_58.903046Z | 16.20KB | FULL | http://localhost:9000 | ffa8024f

- Bucket | Size | Items | Mutations | Tombstones | Views | FTS | Ind
- travel-sample | 16.20KB | 0 | 0 | 0 | 0 | 0 | 1

```

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 つのバケットも含まれており、バケット内のビュー、インデックス、ミューテーション、トゥームストーンなどの数を確認できます。「single」バックアップリポジトリにも 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、データ項目が 0 個であり、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 出力もサポートしています。

増分バックアップ

`cbbakupmgr` の重要な機能の 1 つは、増分バックアップです。既にバックアップされているデータは、再度バックアップする必要はありません。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

第9章 運用

9.2 バックアップ: cbbakupmgr

で backup サブコマンドを再度実行してみます。

```
$ cbbakupmgr backup -a /data/backup -r cluster -c couchbase://127.0.0.1 -u Administrator

Backing up to 2020-03-25T08_41_21.461311Z
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に注意してください。

さらに、info サブコマンドを使用してバックアップアーカイブを一覧表示すると、バックアップアーカイブは次のようになります。Items の数は4であり、そのすべてが Mutations であることがわかります。これらは、2つの変更と2つの追加に対応しています。

```
$ cbbakupmgr info -a /data/backup --all
Name          | UUID                                     | Size      | # Repos |
backup_repo   | 32c97d5f-821a-4284-840b-9ee7cf8733a3 | 86.37MB   | 2       |

* Name        | Size      | # Backups |
* cluster     | 86.35MB   | 2         |
```

```

+ Backup | Size | Type | Source | Cluster
+ 2020-03-25T08_08_11.770436Z | 54.33MB | FULL | http://localhost:9000 | ffa8024f

- Bucket | Size | Items | Mutations | Tombstones | Views | FTS | Ind
- beer-sample | 18.43MB | 7303 | 7303 | 0 | 1 | 0 | 1

- Bucket | Size | Items | Mutations | Tombstones | Views | FTS | Ind
- travel-sample | 35.91MB | 31591 | 31591 | 0 | 0 | 0 | 1

+ Backup | Size | Type | Source | Cluster
+ 2020-03-25T08_41_21.461311Z | 32.02MB | INCR | http://localhost:9000 | ffa8024f

- Bucket | Size | Items | Mutations | Tombstones | Views | FTS | Ind
- beer-sample | 16.00MB | 0 | 0 | 0 | 1 | 0 | 1

- Bucket | Size | Items | Mutations | Tombstones | Views | FTS | Ind
- travel-sample | 16.02MB | 4 | 4 | 0 | 0 | 0 | 1

* Name | Size | # Backups |
* single | 16.20KB | 1 |

+ Backup | Size | Type | Source | Cluster
+ 2020-03-25T08_08_58.903046Z | 16.20KB | FULL | http://localhost:9000 | ffa8024f

- Bucket | Size | Items | Mutations | Tombstones | Views | FTS | Ind
- travel-sample | 16.20KB | 0 | 0 | 0 | 0 | 0 | 1

```

`backup` 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。使用可能なオプションとその使用方法の詳細については、ドキュメントを参照してください。

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。名前を見つけるには、`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確認します。バックアップ名は常にタイムスタンプになります。たとえば、「cluster」バックアップリポジトリから「2016-03-22T102608.933579821-07_00」を復元するとします。これを行うには、以下のコマンドを実行します。

```
$ cbbakupmgr restore -a /data/backup -r cluster \  
-c http://127.0.0.1:8091 -u Administrator -p password \  
--start 2016-03-22T14_00_16.892277632-07_00 \  
--end 2016-03-22T14_00_16.892277632-07_00 --force-updates  
  
(1/1) Restoring backup 2016-03-22T14_00_16.892277632-07_00  
Copied all data in 2s (Avg. 19.96MB/Sec)          38894 items / 39.91MB  
travel-sample          [=====] 100.00%  
beer-sample            [=====] 100.00%  
  
Restore completed successfully
```

上記のコマンドでは、`--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ています。バックアップを1つだけ復元するため、`--start` と `--end` の両方に同じ値を指定します。また、競合解決をスキップするために、`--force-updates` フラグを追加しました。これにより、`cbbakupmgr` は、クラスター上のキーと値のペアが新しく更新されている場合でも、キーと値のペアを強制的に上書きして復元するように指示されます。クラスターで更新した2つの値を見ると、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。

ここでは詳細について触れませんが、`restore` サブコマンドでは、バックアップされた一部のデータ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ので、スペースを再利用できるようにすることが重要。これを行うには、`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します。「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ため、以下のコマンドを使用してこれらのバックアップをマージします。

```
$ cbbakupmgr merge -a /data/backup -r cluster \
  --start oldest --end latest

Merge completed successfully
```

バックアップをマージした後、`list` サブコマンドを使用して、実行したばかりの `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できます。

```
$ cbbakupmgr info -a /data/backup --all
```

Name	UUID	Size	# Repos
backup_repo	32c97d5f-821a-4284-840b-9ee7cf8733a3	54.35MB	2

* Name	Size	# Backups
* cluster	54.33MB	1

+ Backup	Size	Type	Source	Cluster UUID
+ 2020-03-25T08_41_21.461311Z	54.33MB	MERGE - FULL	Merge	ffa8024f899ca6a

- Bucket	Size	Items	Mutations	Tombstones	Views	FTS	Ind
- beer-sample	18.43MB	7303	7303	0	1	0	1

```

-      Bucket          | Size      | Items  | Mutations  | Tombstones  | Views  | FTS  | I
-      travel-sample   | 35.91MB   | 31593  | 31593      | 0           | 0      | 0    | 1

* Name      | Size      | # Backups |
* single    | 16.20KB   | 1         |

+      Backup          | Size      | Type   | Source          | Cluster
+      2020-03-25T08_08_58.903046Z | 16.20KB   | FULL   | http://localhost:9000 | ffa8024f

-      Bucket          | Size      | Items  | Mutations  | Tombstones  | Views  | FTS  | I
-      travel-sample   | 16.20KB   | 0       | 0          | 0           | 0      | 0    | 1

```

「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、マージの最新のバックアップの名前を反映しています。また、`travel-sample` バケットには 31593 個のデータ項目があります。2 番目のバックアップに 2 つの新しいアイテムがあったため、これは元のバックアップより 2 つ多いアイテムです。更新された 2 つのアイテムは、マージ中に重複排除されたため、`info` サブコマンドによって表示されるカウントに追加されません。

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、「クラスター」バックアップリポジトリを削除する方法を示す例です。


```
$ cbbackupmgr remove -a /data/backup -r cluster
```

```
Backup repository 'cluster' deleted successfully from archive '/data/backup'
```

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なくなっていることがわかります。

参考情報

Couchbase 公式ドキュメント `cbbackupmgr` tutorial^{*7}

^{*7} <https://docs.couchbase.com/server/current/backup-restore/cbbackupmgr-tutorial.html>



第 10 章

セキュリティ

10.1 Couchbase Server セキュリティ概観

認証 (Authentication)

Couchbase Server へのユーザ（理者、アプリケーションを含む）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（ネイティブ LDAP、saslauthd、および PAM）。認証には、資格情報をサーバーに直接渡すか、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。接続は、SCRAM と TLS を使用して保護できます。

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、特定のシステムリソースへのアクセスの程度を許可する権限のセットです。これにより、アクセスしようとしているシステムリソース、および実行したい操作に基づいて、アクセスを許可または拒否できます。ロールは、ユーザーに対して割り当てることができます。実行時、ロールは、ユーザ認証を介して、

クライアントに付与されます。

■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、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。

監査 (Auditing)

Couchbase Server で実行されるアクションを監査することを可能にします。アクションのレビューにより、管理者はシステム管理タスクが適切に実行されていることを確認できます。

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エンコードされます。そのため、データを安全に保存または送信できます。これにより、ユーザーデータのプライバシー、およびサーバーとそのクライアントの整合性が保証されます。

通信データ (On the Wire / in motion)

TLS が利用できます (クライアント・サーバー、管理者アクセス、およびセキュア XDCR)。また、TLS のための X.509 certificates の利用をサポートしています。

永続化データ (At Rest)

サードパーティーベンダー (Vormetric, Protegrity, SafeNet) によって提供されます。

JSON ドキュメントのフィールド

SDK を用いた暗号化をサポートしています。

その他

その他セキュリティを管理可能な領域は次のとおりです。

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可能です。クラスターに対するアクセスのみではなく、クラスタを構成するノード間のネットワーク通信の保護も可能です。

証明書

証明書を、クラスターに対して定義・登録できます。加えて、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

ログ中に、個人情報が含まれないように設定することが可能です。

セッション

ユーザーが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

適切なファイヤーウォール設定のために、Couchbase Server が利用する必要がある全てのポート番号の情報が公開されています。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も可能です。

参考情報

Couchbase 公式ドキュメント Security^{*1}

Couchbase 公式ドキュメント Manage Security^{*2}

Couchbase 公式ドキュメント Network and Firewall Requirements^{*3}

Couchbase ブログ Data encryption in Couchbase Server^{*4}

GluuFederation Github リポジトリ Couchbase should not listen by default on all server IPs^{*5}

10.2 認証 Authentication

Couchbase Server にアクセスするには、ユーザーが認証されている必要があります。認証（Authentication）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。認証が成功した後、承認（Authorization）を実行できます。これにより、ユーザーの適切なアクセスレベルが決定されます。

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。

^{*1} <https://docs.couchbase.com/server/current/learn/security/security-overview.html>

^{*2} <https://docs.couchbase.com/server/current/manage/manage-security/security-management-overview.html>

^{*3} <https://docs.couchbase.com/server/current/install/install-ports.html>

^{*4} <https://blog.couchbase.com/data-encryption-in-couchbase-server/>

^{*5} <https://github.com/GluuFederation/community-edition-setup/issues/452#issuecomment-402791618>

これらの情報は、アクセスを試みるユーザー（つまり、管理者またはアプリケーション）からシステムに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 クラスター自体、または外部システム、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

まず、Couchbase クラスターの初期化時に、ユーザーがビルトイン管理者アカウントのユーザー名（デフォルトでは、Administrator）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。追加ユーザーを管理者権限を持つユーザーとして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、追加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 (reset-admin-password サブコマンドによる couchbase-cli) を実行する必要があります。

```
couchbase-cli reset-admin-password [--regenerate] [--new-password <password>]
[--port <port>]
```

外部アクセシビリティ管理

ユーザーは、パスワードを指定する必要のない外部ユーザーとしてクラスターに追加することもできます。外部ユーザーは外部システムで認証されます。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（ライトウェイト・ディレクトリ・アクセス・プロトコル）
- PAM（プラグブル認証モジュール）フレームワーク

■コラム：エディションによる差異

LDAP、PAM を用いた管理は、Couchbase Server の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

Couchbase 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、指定されたユーザー名とパスワードが平文で渡されます。

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が暗号化された形式で渡される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、Simple Authentication and Security Layer (SASL) 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する必要があります。SCRAM メカニズムにより、保護された形式でのみパスワードを送信することにより、アプリケーションは安全に認証できます。SHA ベースのハッシュ関数をサポートするには、ドライバの更新が必要になる場合があります。

このパスワード認証メカニズムには、次の方式があります（強度による昇順）。

- PLAIN：クライアントは暗号化されていない形式でパスワードを送信します。すべてのクライアントがこの認証方法をサポートしています。これは安全ではなく、送信中にパスワードが盗まれるのを防ぐことはできません。
- 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のグループの 1 つである、256 ビットのキーを使用します。
- SCRAM-SHA512：SHA2 グループの別のハッシュ関数である SCRAM-SHA512 は、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。

最初のクライアントサーバーネゴシエーションでは、クライアント（アプリケーション）とサーバー（Couchbase Server）の OS 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。ただし、クライアントが SCRAM-SHA1 プロトコルもサポートしている場合は、SCRAM-SHA1 が使用されます。

チャレンジレスポンス方式は、暗号化されたチャネルと暗号化されていないチャネルの両方を介して送信できます。

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認証することに注意してください。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。

証明書ベースの認証は、認証局（CA）に依存して ID を検証し、証明書を発行します。証明書には、識別したエンティティの名前、有効期限、証明書

を発行した CA の名前、発行した CA のデジタル署名などの情報が含まれています。

SASL

SCRAM は、Salted Challenge Response Authentication Mechanism の略称です。これによって、「burute force」攻撃や、「man in the middle」攻撃を防ぐことができます。

SCRAM SHA 512 が、Couchbase SDK の標準の認証方式です。

パスワードポリシー設定

下記のように CLI を使って、パスワードポリシーを変更できます。

```
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
}
```

参考情報

Couchbase 公式ドキュメント Certificates^{*6}

Couchbase 公式ドキュメント Manage Certificates^{*7}

Couchbase 公式ドキュメント Configureing saslauthd^{*8}

10.3 認可 Authorization

ロールの基本的なユースケース

ロールの基本的なユースケースを考えるに当たって、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、アプリケーション、そして開発者に分類し、それぞれの特徴を整理します。

管理者

管理者は、(Web コンソールにログインして) 管理タスクを実行します。基本的に、データの読み取りまたは書き込みを行う必要はありません。

アプリケーション

アプリケーションは、データの書き込み、および（または）、読み取りを行います。Web コンソールにログインしたり、クラスターの設定を変更する必要はありません。

^{*6} <https://docs.couchbase.com/server/current/learn/security/certificates.html>

^{*7} <https://docs.couchbase.com/server/current/manage/manage-security/manage-certificates.html>

^{*8} <https://docs.couchbase.com/server/current/manage/manage-security/configure-saslauthd.html>

開発者

開発者は、自分の担当するアプリケーションに関係するデータを参照する必要があり、多くの場合データの書き込み権限も必要になります。また、WEB コンソールにログインし、クラスターの情報を参照したり、データを確認・操作することも必要とされるでしょう。

Couchbase Server のロール**Full Admin**

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

Read-only Admin ロールは、Couchbase Web コンソールへのアクセスを許可します。

Application Access

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、リソース（スコープやコレクション、あるいは各サービス）および操作（データに対する読み取り、書き込み等）の両面で、より詳細なロールによるアクセスコントロールを利用可能です。

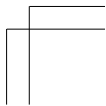
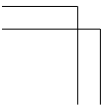
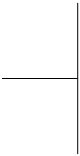
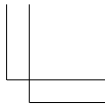
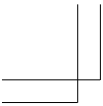
参考情報

Couchbase 公式ドキュメント Roles^{*9}

Couchbase 公式ドキュメント Role-Based Access Control (RBAC)^{*10}

^{*9} <https://docs.couchbase.com/server/current/learn/security/roles.html>

^{*10} <https://docs.couchbase.com/server/current/rest-api/rbac.html>



第 11 章

コマンドライン操作

11.1 コマンド

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。

基本的な構文は以下の通りです。

```
couchbase-cli <コマンド> [オプション]
```

基本オプション

- `- version` バージョン情報を出力します。
- `- help (-h)` ヘルプメッセージを出力します。

コマンド

`couchbase-cli` で利用可能なコマンドを紹介します。詳細な利用方法は公式ドキュメント^{*1}をご覧ください。

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

第 11 章 コマンドライン操作

11.1 コマンド

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	Couchbase データバケットを圧縮します。
bucket-create	新しい Couchbase データバケットを作成します。
bucket-delete	Couchbase データバケットを削除します。
bucket-edit	Couchbase データバケット設定を編集します。
bucket-flush	Couchbase データバケットのデータをフラッシュします。
bucket-list	すべての Couchbase データ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	Couchbase クラスターを初期化します。
ip-family	Couchbase クラスターを初期化します。
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。

11.2 ストレスツール

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) テストは避けて通れない要素です。

Couchbase Server では、Data サービスと、Query サービスという二種類のサービスが、基本的なデータ操作に関係しています。

それぞれのサービスには、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	ストレスツール	データ操作
Data サービス	pillowfight	全て
Query サービス	n1qlback	(主に) READ

Query サービス (N1QL クエリ) は、INSERT 、 DELETE や UPDATE もサポートしていますが、アプリケーション連携において、キーによる操作が可能なところで、クエリを用いることは、アンチパターンとなるため (WHERE 句を用いた条件によるデータ削除・更新や、INSERT .. FROM SELECT... のような処理が必要な要件がある場合は、その限りではありません)、実際的には、READ の処理が中心となります。

これらのツールは、Couchbase Server の C 言語 SDK libcouchbase の一部として提供されます。

インストール

CentOS/RHEL

```
$ sudo yum check-update
$ sudo yum search libcouchbase
```

```
$ sudo yum install libcouchbase3 libcouchbase-devel libcouchbase3-tools
```

Mac

```
$ brew update  
$ brew install libcouchbase
```

Windows

SDK リリースノート^{*2}から、バイナリをダウンロード・解凍して利用します。

その他の OS についてもドキュメント^{*3}を参照してください。

共通オプション

以下のオプションは、クラスターへの接続に関連する、libcouchbase コマンドの共通オプションです。

^{*2} <https://docs.couchbase.com/c-sdk/current/project-docs/sdk-release-notes.html#install-windows>

^{*3} <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	接続のための追加のオプション (例. -D-timeout=[秒数])
--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 秘密鍵へのパス
--dump	なし	操作完了後に詳細な内部状態をダンプする

pillowfight

`pillowfight` 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。

`pillowfight` は、Couchbase C SDK `libcouchbase` の一部として提供されます。

オプション

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	最小ペイロードサイズ [デフォルト: 50]
--max-size	-M	最大ペイロードサイズ [デフォルト: 5120]
--pause-at-end	-E	実行の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: 1]
--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]
--rate-limit	なし	秒あたり操作の制限を設定（スレッド単位） [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-s-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: ALSE]
--noop	なし	ドキュメント操作の代わりに NOOP を使用 [デフォルト: FALSE]
--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: 1]
--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	アイテムを指定した数のノードに永続化するまでブロックする（\"-1\" の場合、
--replicate-to	なし	アイテムを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、
--lock	なし	更新用ロックキーへ時間設定ゼロの場合、ロックされない [デフォルト: 0]
--config-cache	-Z	キャッシュ設定へのパス
--help	-?	ヘルプメッセージを表示する

実行例

20 スレッドで実行。1 スレッド当たり 1 バッチ処理実行。

```
$ cbc-pillowfight -t 20 -B 1
```

50 アイテムのデータセットを使って、100回のイテレーションを行う。

```
$ cbc-pillowfight -c 100 -I 50
```

サイズ 100k の JSON ドキュメントを利用。

```
$ cbc-pillowfight --json -m 100000 -M 100000
```

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
```

接続先、アカウントの指定。JSON ドキュメントを、10 秒後に期限切れにする。

```
$ cbc-pillowfight -U couchbase://localhost/pillow -u Administrator -P
```

参考情報

Couchbase 公式ドキュメント C SDK Release Notes and Archives^{*4}

Couchbase 公式ドキュメント Couchbase C Client 3.2.0 cbc-pillowfight^{*5}

Couchbase ブログ Performance Testing and Load Testing Couchbase with Pillowfight^{*6}

n1qlback

n1qlback は、pillowfight 同様、Couchbase C SDK libcouchbase の一部として提供されます。

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（1 行に 1 クエリ）へのパスを渡す必要があります。クエリは、下記のように、サーバーに送信される実際の HTTP POST 本文の形式（JSON 形式）である必要があります。単純なクエリの場合、ステートメントフィールドのみを設定します。

```
{"statement":"SELECT country FROM 'travel-sample'\"}"}
```

^{*4} <https://docs.couchbase.com/c-sdk/current/project-docs/sdk-release-notes.html>

^{*5} https://docs.couchbase.com/sdk-api/couchbase-c-client-3.2.0/md_doc_cbc-pillowfight.html

^{*6} <https://blog.couchbase.com/performance-testing-load-testing-couchbase-pillowfight/>

```
{"statement":"SELECT country, COUNT(country) FROM 'travel-sample' GRO
```

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=$t
```

処理を実行する前にコマンドにより削除される特別なオプション「n1qlback」が、あります。実行前に特定のクエリを準備する必要があるかどうかを判断できます。

```
{"statement":"SELECT * FROM 'travel-sample' WHERE type=$type LIMIT 10
```

n1qlback を利用する際には、Couchbase Server に必要なリソース（データ、インデックス）が定義されている必要があります（当たり前ですが、ツールが必要になるインデックスを事前に作成するようなような補助機能は備えていないということです）。

オプション

- **-f - queryfile=PATH:**

JSON 形式で実行するクエリ本文を含むファイルへのパス。1 行に 1 つのクエリ。

- **-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。各スレッドには、それぞれクライアントオブジェクトが割り当てられます。

- `-e, - error-log=PATH`: ファイルへのパスを指定します。そのファイルへコマンドは失敗したクエリとエラーの詳細を書き込みます。このオプションの使用により、ERRORS メトリックがゼロでない理由を把握することができます。

次のオプションは、`couchbase://host/bucket? option1=value1& option2=value2` のような、URI スタイルのクエリパラメータ（例）として、接続文字列に含めることができます。あるいは、個々のキー=値のペアとして D スイッチと共に利用することができます（-D は内部で接続文字列を作成し、コマンドラインでオプションを簡単に渡すための便宜のために提供されています）。

- `operation_timeout=SECONDS`:

操作タイムアウトを秒単位で指定します。これは、タイムアウトするまでに、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 です。

- `config_cache=PATH`: クライアントがブートストラップ操作によって接続するのではなく、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、クライアントは最初にクラスターに接続し、次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`: サーバーの SSL 証明書へのパス。これは通常、証明書がシステムの openssl インストールにすでに追加されていない限り、SSL 接続に必要です（`couchbases://スキーム`でのみ適用可能）
- `ssl=no_verify`: SSL の証明書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）。これは、SSL 機能をすばやくデバッグするためにのみ使用してください。
- `sasl_mech_force=MECHANISM`: 初期接続を実行するときに、特定の SASL メカニズムを強制的に使用します。これは、デバッグ目的でのみ変更する必要があります。現在サポートされているメカニズムは、

PLAIN と CRAM-MD5 です。

実行例

以下は、3つのクエリが5つのスレッドで実行されるファイルを作成します。また、クエリに必要なインデックスの作成を事前に行っています。

```
cbc n1ql -U couchbase://192.168.72.101/a_bucket 'CREATE INDEX ix_name
cbc n1ql -U couchbase://192.168.72.101/a_bucket 'CREATE INDEX ix_coun

cat queries.txt <<EOF
{"statement":"SELECT country FROM travel-sample WHERE travel-sample.c
{"statement":"SELECT name FROM travel-sample LIMIT 10"}
{"statement":"SELECT country, COUNT(country) FROM travel-sample GROU
EOF

cbc-n1qlback -U couchbase://192.168.72.101/a_bucket -t 5 -f queries.t
```

参考情報

Couchbase 公式ドキュメント Install and Start Using the C SDK with Couchbase Server^{*7}

Couchbase Github リポジトリ cbc-n1qlback^{*8}

^{*7} <https://docs.couchbase.com/c-sdk/current/hello-world/start-using-sdk.html>

^{*8} <https://github.com/couchbase/libcouchbase/blob/master/doc/cbc-n1qlback.markdown>

第 12 章

Couchbase Server の構成要素

12.1 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。ここで扱われるデータは、メモリ、あるいはディスク上に存在します。Data サービスへのメモリの割り当ては設定により調整することが可能です。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。

Couchbase Server は、ドキュメント指向型に分類される NoSQL でありながら、その最も基本的な部分では、純粋な KVS（キーバリューストア）として、設計されています（バリューとして JSON ドキュメントを格納することによって、Query サービス等の JSON 関連機能を活用することができますが、バリューには、バイナリデータを格納することも可能です）。

構成

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 (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とり、応答を提供します。
- 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** 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。これには、パーティションハッシュテーブルが含まれています。これにより、メモリ内およびディスク上の、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ます。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（揮発性のエフェメラル型バケットではなく、デフォルトの Couchbase 型バケットの場合）ディスクキューを介して、ディスクに書き込まれます。
- **チェックポイントマネージャ** チェックポイントマネージャは、チェックポイント (checkpoint) という名称のデータ (構造) を使用して、ア

アイテムの変更を追跡します。チェックポイントには、メモリ内のアイテムとして加えられ、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変更が記録されます。

- **アイテムページャー** 必要に応じてスペースを解放するために、最近使用されていないアイテムをメモリから取り除きます。
- **フラッシャー** バケット内のすべてのアイテムを一括削除します。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) ページャー** 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。その後、トゥームストーンは、デフォルトの 3 日間残ります。有効期限ページャーは、デフォルトで 60 分ごとに実行されます。
- **バッチリーダー** 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

主に I/O を処理するためのスレッドのプールです。スレッドは 4 種類に分けられ、互いに独立して、影響を与えずに実行されます。

- **Non IO** ディスクアクセスを必要としないスケジューラー専用のタスクです。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が含まれます。
- **Aux IO (補助 auxiliary IO)** フェッチ、スキャン、およびバックフィルタスク等、補助的な処理を担います。
- **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。

12.2 Query サービス

Couchbase Server は、ドキュメント指向型に分類される NoSQL でありながら、N1QL という SQL を JSON に対して拡張したクエリ言語を用いて、JSON データに対するクエリを実行することができます。

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。

構成

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

有効なステートメントであるかどうかを判断するために、受信されたクエリにパーサーを適用します。また、利用可能な実行パスを評価する**オブティマイザー**を用いて、待ち時間が最も短いパスを決定します。さらに、決定された待ち時間が最も短いパスを使用によっ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。**実行エンジン**は、オペレータを受け取り、それを実行します。

データストア

データソースへのアクセスを提供します。Couchbase Server ストアが、Couchbase Server のデータ、インデックス、認証情報を処理するために使用されます。また、他のデータストアとして、ローカルファイルシステム等も含まれます。

クエリの実行

Query サービス内部の処理、及び他のサービスとの連携の内容は、以下の通りです。

- クエリプロセッサは、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。
- 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。
- 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。
- Query サービスは、Filter、Aggregate、Sort などの追加の処理を実行します。

可能な場合、処理は並行で実行されます。

12.3 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。Data サービスで生じるデータの変更に際して、DCP(データチェンジプロトコル) を介して、データの変更を反映し、インデックスが更新されます。

構成

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対応

プロジェクターとルーター

プロジェクターとルーターは Data サービス・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。インデックスが作成されていない場合、プロジェクターとルーターは何もしません。

以下は、Index サービスのスーパーバイザーと Data サービスのプロジェクター・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。

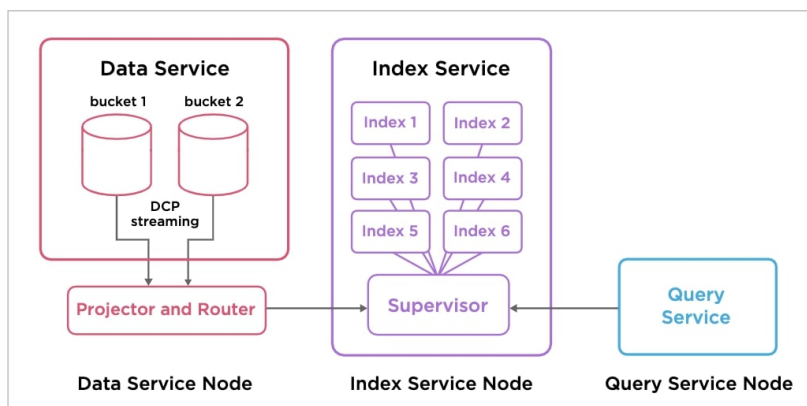


図 12.1: services relationship

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

他サービスとの連携

インデックス作成時

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。インデックスの初回作成時、スーパーバイザーがインデックス定義をプロジェクターとルーターに渡します。

データ更新時

プロジェクターとルーターは、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テーション（更新）情報が、インデックスフィールドへのミューテーションを含

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

む場合には、プロジェクターとルーターは、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。

インデックス利用時

Query サービスから、スーパーバイザーに対して、リクエストが送られ、スーパーバイザーが Query サービスへインデックス情報を返します。

ストレージ

Index サービスは、標準ストレージまたはメモリ最適化ストレージのいずれかを使用するように構成できます。

標準ストレージ

個々のインデックスに加えられたすべての変更をディスクに保持します。各インデックスは専用のファイルからなります。

- コミュニティでは、インデックス作成時に、追加書き込み専用 (append-only) モード、あるいは循環再利用 (circular-reuse) モードの何かを選択します。
- エンタープライズエディションでは、自動的に圧縮処理されます。

メモリ最適化ストレージ

EnterpriseEdition でのみ使用できます。インデックスはメモリに保存されます。これにより、メンテナンス、スキャン、および変更の効率が向上します。インデックスのスナップショットはディスク上に保持され、ノード障害が発生した場合に迅速に回復できるようにします。メモリ最適化ストレージには、メモリ消費の最適化のため、skiplist (B ツリー) 構造が使用されています。

12.4 クラスターマネージャー

クラスターマネージャーは、クラスタを構成する全てのノード上に存在します。他の分散アーキテクチャーで見られるマスター/スレーブ（マスター/ワーカー）構成のような、管理するノードと管理されるノードからなる構成とは異なっています。（後に詳しく触れますが）全てのノードに存在するクラスターマネージャー（インスタンス）の中から、マスターサービスとしてのクラスターマネージャー（インスタンス）が選出されます。

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

ns-server

クラスターマネージャーの中心となるプロセスは ns-server であり、その構成は次のとおりです。

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、クラスタ全体に影響を与える運用を管理します。常に、クラスタ上のノードの 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もあります。

REST 管理 (UI および CLI)

REST API による管理機能を提供します。これは、Couchbase Web コンソールによって提供されるユーザーインターフェイスと Couchbase 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。

第 12 章 Couchbase Server の構成要素 12.4 クラスタマネージャー

認証 (Authentication)

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。各ロールはさまざまなシステム権限に関連付けられています。資格情報（ユーザー名とパスワード）に基づいて、ロールが割り当てられます。

babysitter

babysitter は、CouchbaseServer の様々なプロセスの保守を担当します。（保守の対象には、クラスタマネージャーのもう一つのプロセスである ns-server を含みます。）

- 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。

babysitter の位置づけとして特徴的なのは、babysitter は、クラスタの概念とは独立しており（クラスタの一部であることが意識される役割を持っているわけではなく）、純粹に、そのノードで稼働しているサービスの保守を行っているということです。

マスターサービスの役割

マスターサービスの役割を大別すると下記があります。

- **サービス管理** ノードの現在の状態を管理し、その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を管理します。レプリケーション、フェイルオーバー、再起動、および統計収集をサポートします。

また、以下のような面についても、マスターサービスの役割として理解しておくことは重要です。

- **分散処理機能一般** ノードの検出、構成変更に関するメッセージングとアラート、レプリケーション、ハートビートの送信をサポートします。
- **ローカル機能一般** ローカルに提供される機能（ライブラリ、ワークキュー、ロギング、クロック、ID、およびイベント）をサポートします。

クラスタメンバー管理

選出されたマスターサービスは、クラスタのメンバー構成に責任があります。クラスタのトポロジが変更されると、既存のワークロードの処理を継続しながら、一連の操作が実行され、クラスタの再構成が行われます。

ノードの追加

1. マスターサービスは、リバランスを開始し、vBucket マップを再計算します。
2. データを受信するノードは、各 vBucket の既存のノードから DCP レプリケーションストリームを開始し、それらの vBucket の新しいコピーの作成を開始します。これは、新しい vBucket マップレイアウトに応じて、アクティブ vBucket とレプリカ vBucket の両方で発生します。
3. 新しいアクティブ vBucket でデータが更新されるたびに、そのデータは複製され、関係するインデックスが更新されます。
4. 新しい vBucket の更新が完了すると、古い vBucket から新しい vBucket へのスイッチオーバーが発生します。
5. 新しいノード上の新しい vBucket がアクティブになると、マスターサービスは、新しい vBucket マップがすべてのノードとクライアントに通知されるように通知します。

第 12 章 Couchbase Server の構成要素 12.4 クラスタマネージャー

6. 以上のプロセスは、リバランスが完了するまで繰り返されます。

ノードの削除

Data サービスノードを削除するプロセスは、追加するプロセスと似ています。維持されるノードで新たに vBuckets が作成され、削除されるノードの vBuckets からデータが移動されます。削除対象ノードから、vBucket 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。

Data サービスをホストしないノードを追加または削除する場合、データは移動されません。したがって、ノードはデータ移管なしでクラスターに追加またはクラスターから削除されます。

ノード障害の検出

Couchbase Server クラスタ内のノードは、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タ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、自動フェイルオーバーを行います（有効化されており実行可能な条件が整っている場合）。

バケット管理

Couchbase Server バケットは、物理的に 1024 個のマスター vBucket ファイルと、レプリカ vBucket ファイルからなります。マスターサービスは可用性とリバランスのパフォーマンスを最大化するために、これらの vBuckets の配置をコントロールします。vBucket の配置は、クラスタトポロジーが

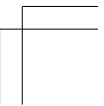
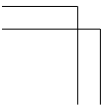
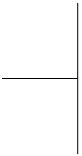
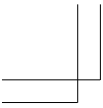
第 12 章 Couchbase Server の構成要素 12.4 クラスターマネージャー

変更されるたびに、次のルールによって再編成されます。

- マスターとレプリカの 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。
- マスター vBucket に対してサーバーグループが定義されている場合、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プッシュされます。



第 13 章

構成管理

13.1 コンパクション

メカニズム

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。この結果として (特にデータが削除された場合に) 生じるデータファイル内のギャップは、コンパクションと呼ばれるプロセスを通じて解消されます。

コンパクションの際には、新しいファイルが作成されます。その間、既存のデータベース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、古いデータベースファイルが無効になります。その後、新しく作成されたデータベースファイルで受信更新が続行されます。その後、古いデータベースファイルがシステムから削除されます。

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。

運用における留意点

コンパクションの運用にあたり、下記の要因に注意を払う必要があります。

- 余裕のあるリソース状況

コンパクションプロセスはディスクと CPU の両方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するため、コンパクションを行うために現在のデータベースファイルのディスク領域の 2 倍が必要になる場合があります。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ベース・サイズの 2 倍未満の場合、コンパクション処理は行われず、ログに警告が出されます。

オート・コンパクション

オート・コンパクションのトリガー

オート・コンパクションのトリガーには、下記の 2 つの要素があります。

- データベースの断片化 これがプライマリ設定です。この数値を

10% に設定すると、データファイルが 10% 断片化された時点でオート・コンパクションがトリガーされます (オート・コンパクションの時間帯制限がない限り)

- 時間帯データベースが多用されているときにオート・コンパクションが行われないようにするには、オート・コンパクションが許可される時間帯を設定します。これは、時・分の単位で設定できます (たとえば、01:00 から 06:00 の間に実行)。デフォルトでは、コンパクション

は順次実行されます。

メタデータのページ間隔

オート・コンパクションに関連深い、別の設定項目として、メタデータのページ間隔があります。期限切れアイテムおよび削除済みアイテムの Tombstones（削除済みアイテムと期限切れアイテムのキーとメタデータを含むレコード）は、オート・コンパクションプロセスの一部として削除されます。

オート・コンパクション設定

オート・コンパクションの設定は、クラスター単位とバケット単位で行うことができます。下記は、設定画面のイメージです。上記解説の内容を参考に、ご覧になってください。

Auto-Compaction ⓘ

☒ Override the default auto-compaction settings?

Database Fragmentation

Set the database fragmentation level to determine the point when compaction is triggered.

☒ %

☐ MB

View Fragmentation

Set the view fragmentation level to determine the point when compaction is triggered.

☒ %

☐ MB

Time Interval

☐ Set the time interval for when compaction is allowed to run

Start Time

:

End Time

:

☐ Abort compaction exceeding the set time interval

☐ Compact buckets and views indexes in parallel

NOTE FOR GSI INDEXES: Auto-compaction settings are unnecessary for memory-optimized and plasma-based indexes.

Metadata Purge Interval ⓘ

For Couchbase buckets ONLY.

days

Range .04 (1hr) - 60 days

図 13.1: compaction settings

マニュアル実行

REST API コールを用いて、コンパクションをマニュアルで実行することも可能です。

参考情報

Couchbase 公式ドキュメント Compaction API^{*1}

Couchbase 公式ドキュメント Performing Compaction Manually^{*2}

13.2 IP アドレスのバインド

Couchbase Server において、IP アドレスのバインディングがどのように対処されているかを見ていきます。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あります。

クラスターマップとの関係

Couchbase Server では、クライアント/アプリケーションが SDK を介して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、アプリケーションに設定されているノード（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、以降の接続は、対象となるノードに対して直接行われます。つまり、ブートストラップ接続先への一方方向のリクエストだけを考えて、IP やドメイン名の解決について外部で整合性が取れていれば良いという訳にはいかない、ということです。

^{*1} <https://docs.couchbase.com/server/current/rest-api/compaction-rest-api.html>

^{*2} <https://docs.couchbase.com/server/current/rest-api/rest-compact-post.html>

代替 (Alternate) アドレス

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」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ができます)。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。代替アドレスの設定は必要に応じて行われます。

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は、意識しておく価値があるでしょう。

代替アドレスが使われる 2 つのパターンを以下に示します。

外部ネットワークからの接続

まず注意しておく、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ではないことにご留意ください。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。

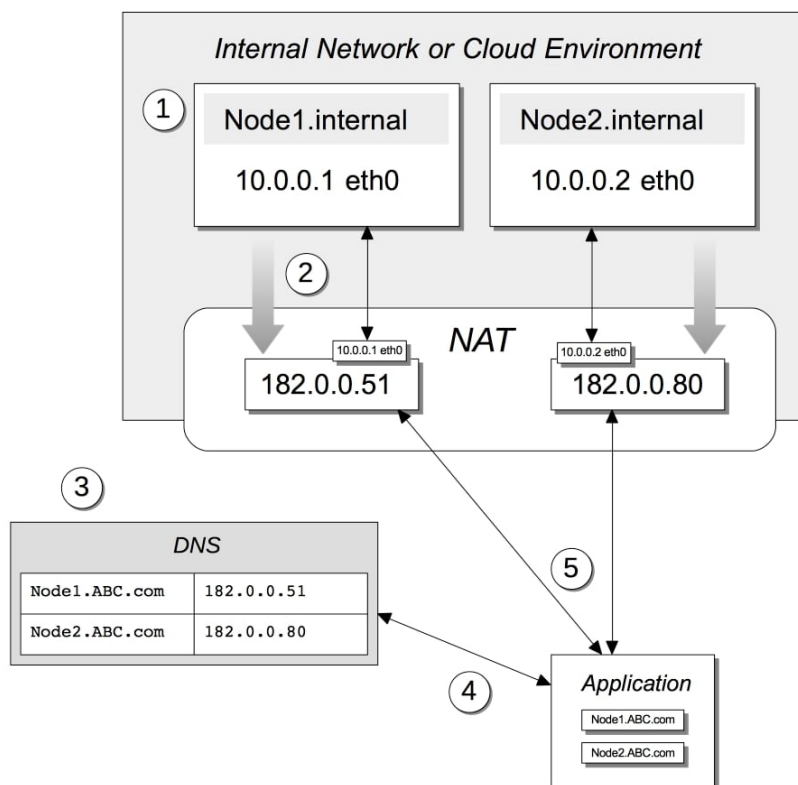


図 13.2: nat

このケースでは、クラスターの構築時（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。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、それらを代替アドレスに割り当てます。

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。

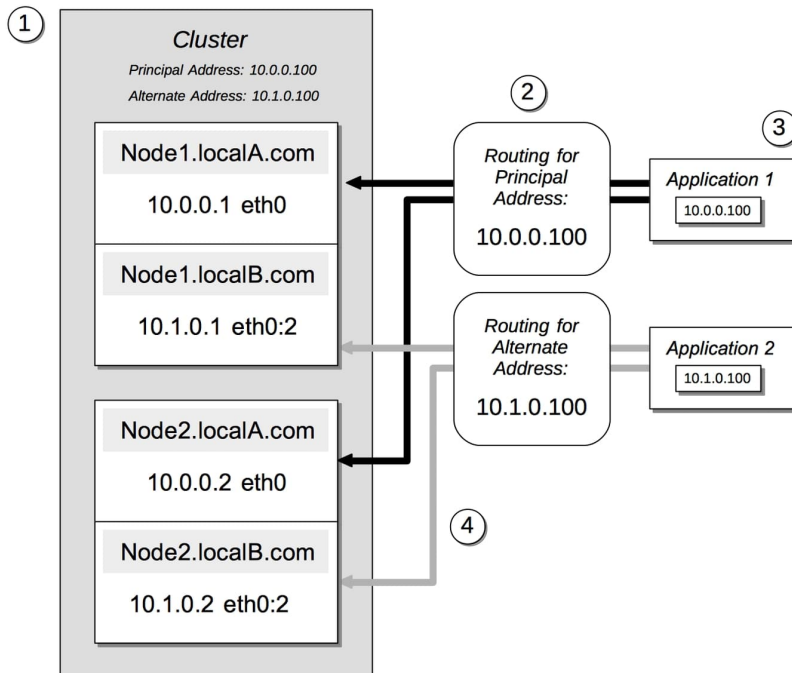


図 13.3: dual network

ここでは、2つのサーバーがあり、それぞれプライマリ IP およびセカンダリ IP 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、それぞれプリンシパルアドレス、代替アドレスのいずれかのみを利用します。

代替アドレスの割り当て方法

次の 2 つの方法で代替アドレスを割り当てることができます。ここでは、いくつかのサンプルを例示します。詳細については、公式ドキュメントをご覧ください。

CLI

以下が、基本となるシノプシスです。

```
couchbase-cli setting-alternate-address [--cluster <url>] [--username <username>] [--password <password>] [--list] [--set] [--remove] [--hostname <hostname>] [--ports <ports>]
```

以下は、`--list` パラメータを用いて、現在の代替アドレスと現在のすべてのポート番号設定を表示する例です。

```
$ couchbase-cli setting-alternate-address -c 192.168.1.5:8091 --username <username> --password password --list
```

REST API

以下は、代替アドレス 10.10.10.11 をノード 10.143.192.101 に割り当て、それに加えて、代替ポート番号 9000 と 9050 を、それぞれデータおよ

びクエリサービスへ割り当てる例です。

```
curl -v -X PUT -u Administrator:password \  
http://10.143.192.101:8091/node/controller/setupAlternateAddresses/external \  
-d hostname=10.10.10.11 \  
-d kv=9000 \  
-d n1ql=9050
```

参考情報

Couchbase 公式ドキュメント Alternate Addresses^{*3}

Couchbase 公式ドキュメント REST API リファレンス Managing Alternate Addresses^{*4}

Couchbase 公式ドキュメント CLI リファレンス setting-alternate-address^{*5}

^{*3} <https://docs.couchbase.com/server/current/learn/clusters-and-availability/connectivity.html#alternate-addresses>

^{*4} <https://docs.couchbase.com/server/current/rest-api/rest-set-up-alternate-address.html>

^{*5} <https://docs.couchbase.com/server/current/cli/cbcli/couchbase-cli-setting-alternate-address.html>

第 14 章

外部システム連携

14.1 コネクタ

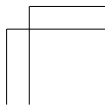
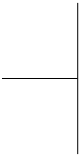
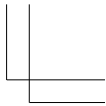
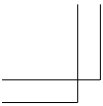
Couchbase Server と以下の外部システムを連携するためのコネクタが提供されています。

- Elasticsearch
- Kafka
- Spark

参考情報

Couchbase 公式ドキュメント Connector Guides^{*1}

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



第 15 章

Node.js アプリケーション 開発

15.1 なぜ、JavaScript と NoSQL の組み合わせ なのか？

NoSQL は、非 RDB (RDB 以降の技術) であることのみを共通点として、様々な異なる特色を持った技術に対する総称となっていますが、ここで対象とするカテゴリーは、その中でもドキュメント指向データベース (ドキュメントストア) と呼ばれるものです。

ドキュメントストアは、JSON データを格納することを特徴としており、自ずと、JavaScript と親和性を持っています (JSON = Java Script Object Notation)。

その一方、RDB から移行するだけの利点があるのか、というのが実際的な関心なのではないかと思います。こちらについては後ほど触れていきます。

15.2 NoSQL/Couchbase を選択する理由

Couchbase なら、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

これは、RDB の欠点と見ることもできます。つまり...

- アプリケーションが必要とするデータ構造（ドメインオブジェクト）と、RDB が要請する形式（第一正規形テーブル構造）との間には、断絶がある。
- アプリケーションの設計、実装、改善、機能追加など、全ての工程において、データベースとの兼ね合いを図る必要がある（密結合）

これに対して、データ層が、JSON を許容した場合...

- データ層は、第一正規形を要請しないため、アプリケーションが必要とするデータ構造（ドメインオブジェクト）そのものを格納することができる。
- (JSON には、データ構造に関する情報がデータ自体に含まれているため) アプリケーション設計・開発工程において、特にデータ設計の変化に（データベース側の作業を伴うことなく）柔軟に対応できる

クエリ言語の利点

これは、RDB の持つ大きな利点であり、標準化されたクエリ言語 (SQL) が様々な異なるデータベースで利用できることは、技術者層の拡大に繋がります。SQL の習得は、(特にオープンシステムの WEB アプリケーション全盛時代には) システム開発者にとって、必須知識といえるものとなっています

した。

Couchbase Server を選択することで、開発者は、SQL の知識を活用することができます。

15.3 キーバリューストア操作

他の箇所でも、N1QL によるクエリについて RDB との相違点を中心に解説しました。Couchbase Server では、クエリが唯一のインターフェイスではなく、キーバリューストア同様にキー指定によるドキュメントの操作を行うことが可能です。

CRUD 操作

キーバリューストアの基本的な操作について、プログラミング例を参考として示します。詳細についてはドキュメント^{*1}を参照ください。

CREATE

```
const result = await collection.insert(key, document);
```

READ

```
const result = await collection.get(key);  
document = result.value;
```

^{*1} <https://docs.couchbase.com/nodejs-sdk/current/howtos/kv-operations.html>

UPDATE

```
let result = await collection.replace(key, document);
```

UPSERT

```
let result = await collection.upsert(key, document);
```

DELETE

```
const result = await collection.remove(key);
```

サブドキュメント操作

純粋なキーバリューストアと異なり、JSON データベースを利用している場合、アプリケーションは JSON ドキュメントの特定の箇所のみを取得すれば十分な場合があります。あるいは、アプリケーションでデータの更新を行う場合に、ドキュメント全体を取得し、その一部を編集した後に、ドキュメントをデータベースに対して更新するのではなく、データベースに対してドキュメントの指定箇所の更新を指示できることが重要です。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、それは唯一の方法ではありません。Couchbase Server では、Data サービス API を用いて、JSON データの一部の取得・更新を行うことができます。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について、プログラミング例を参考として示

します。詳細についてはドキュメント^{*2}を参照ください。

READ

```
var result = await collection.lookupIn("customer123", [
  couchbase.LookupInSpec.get("addresses.delivery.country"),
]);
var country = result.content[0].value; //'United Kingdom'
```

EXISTS

サブドキュメントを実際に取得しなくても、指定したパスのサブドキュメントを存在するかどうかを確認することが可能です。

```
var result = await collection.lookupIn("customer123", [
  couchbase.LookupInSpec.exists("purchases.pending[-1]"),
]);
console.log("Path exists? ", result.content[0].value);
```

複数操作の同時実行

```
var result = await collection.lookupIn("customer123", [
  couchbase.LookupInSpec.get("addresses.delivery.country"),
  couchbase.LookupInSpec.exists("purchases.pending[-1]"),
]);
```

^{*2} <https://docs.couchbase.com/nodejs-sdk/current/howtos/subdocument-operations.html>

```
]);  
  
console.log(result.content[0].value);  
console.log("Path exists?", result.content[1].value);
```

CREATE

```
await collection.mutateIn("customer123", [  
  couchbase.MutateInSpec.insert("purchases.complete", [42, true, "None"]),  
]);
```

UPDATE

```
await collection.mutateIn("customer123", [  
  couchbase.MutateInSpec.replace("email", "dougr96@hotmail.com"),  
]);
```

UPSERT

```
await collection.mutateIn("customer123", [  
  couchbase.MutateInSpec.upsert("fax", "311-555-0151"),  
]);
```

DELETE

第 15 章 Node.js アプリケーション開発15.4 データー貫性 (Consistency)

```
await collection.mutateIn("customer123", [
  couchbase.MutateInSpec.remove("addresses.billing"),
]);
```

15.4 データー貫性 (Consistency)

分散アーキテクチャーにおけるインデックス

Couchbase Server と RDBMS との違いとして、メモリーファーストアーキテクチャーがあります。メモリーファーストアーキテクチャーは、永続化装置および複製（レプリケ）の作成が、アプリケーション（への応答）性能のボトルネックになることを回避するための仕組みです。そのための仕組みとして、Couchbase Server は、ノード内部でディスクへの書き込みのためのキューを持っています。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) と呼ばれる仕組みを持っています。この DCP は、レプリカのためのみではなく、クラスター内部でデータの変更を他のノードやサービスに反映するために広く利用されます。（Data サービスにおける）データ変更のインデックスへの反映もその一つです。

インデックス更新から見る RDB との違い

Couchbase Server と RDB との違いをこの観点から見るができます。データを更新する際には、関係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。ただし、RDB では、ユーザーが、いつどのように、データへアクセスしたとしても、データの一貫性が保たれていることを保証するため、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、インデックスの更新が完了した後になります。これは、データー貫性の保証という要件面に加

第 15 章 Node.js アプリケーション開発15.4 データー貫性 (Consistency)

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、技術的な面から見ても自然な挙動と言えるかもしれません。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おり、異なる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、DCP プロトコルを介した、キュー（非同期）のメカニズムで実現されています。

インデックス更新が非同期であることの影響範囲

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アクセスが、一次的なインターフェースであるのに対して、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（たとえば、ある新しい更新が、永続化されておらず、ディスクキューに入っている状態だとしても、後続のクライアントのアクセスへは、メモリ内の最新の情報から応答されます）。さらに、このようなインターフェースの違いを考慮せず、純粋に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、データの（インデックスへの）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、データの同期が行わ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、RDB の開発経験が豊富な方ほど、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。まず、ここではインデックスは、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、データがインデックスに同期さ

第 15 章 Node.js アプリケーション開発15.4 データー貫性 (Consistency)

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）という現象は決して起こりません。そのことを考えれば、影響範囲は非常に限定的であることがわかると思います。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 DATA (自分自身のデータを読む)

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それぞれ別の更新と検索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うると考えると、原因がデータベース内の遅延によるもの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と考えた場合、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。唯一の例外は、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に対して矛盾した結果が返されるケースです。自分で行った更新が、その後の検索結果に反映されていないとしたら（例えばユーザー管理アプリケーション上で、ユーザーを追加した後に、その新規ユーザーが全ユーザーリストに表示されないとしたら）、これは明らかに問題です。

第 15 章 Node.js アプリケーション開発15.4 データー貫性 (Consistency)

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、複製作成、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作成については、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。

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、インデックス更新状況に対して、リクエストの挙動を変えることのできる複数のオプションがあります。

次のオプションを使用できます。

- **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、古い結果が返されることがあります。
- **at_plus**：インデックスが**最新の更新のタイムスタンプ**まで更新されている事を保証します。インデックスの更新に遅延がある場合は、更新を待ちます。
- **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**まで更新されている事を保証して、クエリを実行します。インデックスの更新状況に遅延がある場合は、更新を待ちます。

デフォルトの整合性は **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、

第 15 章 Node.js アプリケーション開発5.5 サンプルアプリケーション紹介

Node.js SDK/JavaScript での上記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,  
  });
```

参考情報

Couchbase 公式ドキュメント Index Consistency^{*3}

15.5 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_step-by-step_node_jp

15.6 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 (Web アプリケーション・フレームワーク)
- A: Angular (フロントエンド・フレームワーク)

^{*3} <https://docs.couchbase.com/java-sdk/current/concept-docs/n1ql-query.html#index-consistency>

- N: Node.js (サーバサイド JavaScript 実行環境)

類似のものとして、MEAN スタックという言葉聞いたことがある方もいるのではないかと思います。この場合の M は、MongoDB を指します。MongoDB は、Couchbase 同様、JSON データを扱う NoSQL データベースです。

15.7 アプリケーション概要

サンプルアプリケーションは、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を必要としないであろう、最小限の機能からなるシンプルなものとなっています。

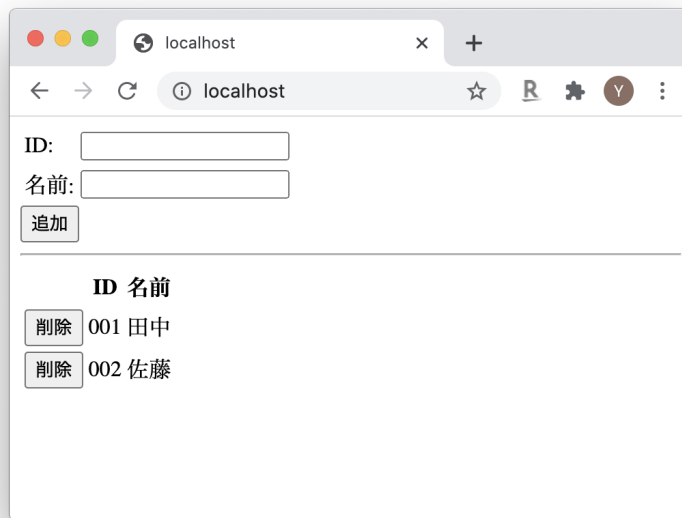


図 15.1: sample app screen

アプリケーション利用方法

Couchbase Server 事前準備

このアプリケーションでは、バケット名として `node_app` を使っています。バケットは事前に作成されている必要があります。

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。

```
CREATE INDEX idx_node_app_user ON node_app.scp.user;
```

コードの取得

上記のリポジトリから、コードを取得します。

```
$ git clone https://github.com/YoshiyukiKono/couchbase_step-by-step_node_jp.git
$ cd couchbase_step-by-step_node_jp
```

プログラム依存関係

その中に含まれる、`package.json` には、下記の依存関係が定義されています。

```
"dependencies": {
  "couchbase": "^3.1.0",
  "express": "^4.17.1"
}
```

実行

このアプリケーションでは、実行の際に、下記のように、`server.js` を使います。

```
$ 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

ユーザーリスト表示

routes.js を見ると、画面のリスト表示のために、下記のようなクエリが使われているのが分かります。

```
const qs = "SELECT name, id from node_app.scp.user";
```

上記の (変数`$qs` に代入された) をクエリ実行している箇所は以下の通りです。

```
const selectUsers = async (key) => {  
  const result = await cluster.query(qs, {  
    scanConsistency: couchbase.QueryScanConsistency.RequestPlus,  
  });  
  return result.rows;  
}
```

script.js 中で、上記クエリのリクエストの戻り値は (`response.data`) は、下記のように、そのまま変数 (`$scope.users`) に代入されます。、index.html 中で、画面表示に利用されます。

```
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);
    });
};
```

代入された変数 (`$scope.users`) は、`index.html` で、画面表示に利用されます。

```
<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` 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、下記のようなものです。

```
const user = {
  type: "user",
  id: req.body.id,
  name: req.body.name,
};
upsertDocument(user);
```

JavaScript のディクショナリをそのまま格納しているのが分かります。

第 15 章 Node.js アプリの ODM(オブジェクトデータモデラー): Ottoman.js

```
const upsertDocument = async (doc) => {
  try {
    const key = `${doc.type}_${doc.id}`;
    const result = await collection.upsert(key, doc);
  } catch (error) {
    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 = `user_${id}`;
    const result = await collection.remove(key);
  } catch (error) {
    console.error(error);
  }
};
```

15.8 ODM(オブジェクトデータモデラー): Ottoman.js

Ottoman.js^{*4}は、Couchbase Server と Node.js のための ODM(オブジェクトデータモデラー) です。Node.js アプリケーション開発に Couchbase

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

第 15 章 Node.js アプリケーション(オブジェクトデータモデラー): Ottoman.js

Server を利用する際、Ottoman の利用は必ずしも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。Ottoman は、MongoDB における Mongoose ODM^{*5}に相当するものであると言えます。

^{*5} <https://mongoosejs.com/>

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、下記から入手可能です。

NoSQL Performance Benchmark 2018: Couchbase Server v5.5, DataStax Enterprise v6 (Cassandra), and MongoDB v3.^{*2}

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

^{*2} <https://www.altoros.com/research-papers/nosql-performance-benchmark-2018-couchbase-server-v5-5-datastax-enterprise-v6-cassandra-and-mongodb-v3-6/>

付録 A NoSQL 性能評価：MongoDB, Cassandra との比較ベンチマーク概要

また、必要と思われる箇所で、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についての指摘を行います。

A.1 ベンチマーク概要

評価基準・ツール

Yahoo! Cloud Serving Benchmark(YCSB)*3

ワークロード・カテゴリー

YCSB 標準

- ワークロード A: 更新処理
- ワークロード E: レンジスキャン

クエリ

- 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。

- 財務: フィルター処理されたトランザクションを一覧表示するためのサーバー側のページネーション
- e コマース: 顧客が利用するさまざまな製品やサービスに関する一連のレポート

*3 <https://en.wikipedia.org/wiki/YCSB>

付録 A NoSQL 性能評価 : MongoDB, Cassandra と Redis の比較ベンチマーク概要

環境

サイズの異なる以下の3通りのクラスターで検証。

- 4 ノード
- 10 ノード
- 20 ノード

データベース用インスタンス

Table 2.1 A detailed description of the Amazon EC2 instance the clusters were deployed to

Family	Storage optimized
Type	i3.2xlarge
vCPUs	8
Memory (GiB)	61
Instance storage (GB)	1 × 1,900 (SSD)
EBS-optimized available	Yes
Network performance	Up to 10 GB

図 A.1: altros nosql benchmark server settings 1 of 2

Platform	64-bit
Operating system	Ubuntu 16.04 LTS

図 A.2: altros nosql benchmark server settings 2 of 2

クライアント用インスタンス

付録 A NoSQL 性能評価：MongoDB, Cassandra との比較ベンチマーク概要

Table 2.2 A detailed description of the Amazon EC2 instance the YCSB client was deployed to

Family	Compute optimized
Type	c4.4xlarge
vCPUs	8
Memory (GiB)	15
EBS-optimized available	Yes
Network performance	High
Platform	64-bit
Operating system	Ubuntu 16.04 LTS

図 A.3: altros nosql benchmark client settings

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
- Durability（耐久性）オプション未使用
- 各データセットに対して1つのレプリカ（複製）

■コラム:

Durability（耐久性）オプションとは、WRITE 処理の際のデータの耐久性のレベルをデフォルトよりも高く設定することです。このオプションを用いた場合、レプリカの作成やディスクへの同期のために、キューを介した非同期処理を行うのではなく、レプリカ作成やディスクへの永続化の完了をもってデータ更新処理完了とされます。

データベース固有の構成

Couchbase Server

クラスタのサイズに関係なく、各ノードは Data サービス、Index サービス、および Query サービスで構成。バケットには、使用可能な RAM(36,178 MB) の 60% を割り当て。Index サービスには、使用可能な RAM の約 40% (約 24GB) を割り当て。作成された各インデックスは、すべての Index サービス (ノード) に複製。

MongoDB

MongoDB は、Router プロセス、Config サーバー、およびデータシャーードからなる階層型クラスタポロジを採用しています。各クラスタサイズ (4、10、および 20 ノード) に対して、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 (クラスタのノード数としてはカウントしない、別のコンピュータ)
- 各シャーードは、3 つのメンバー (Primary、1 つの Secondary、1 つの Arbeiter) からなるレプリカ セットとして構成
- 各クライアントに 3 つの mongos (Router プロセス) をデプロイ

Cassandra

下記の設定を適用

付録 A NoSQL 性能評価：MongoDB, CassandraA2の比較ワークロード別詳細

Table 2.4 The changes applied to each node on each cluster

cassandra.yaml	
memtable_heap_space_in_mb	16,384
memtable_cleanup_threshold	0.11
memtable_flush_writers	40
row_cache_size_in_mb	20,280
commitlog_total_space_in_mb	1,969
cdc_total_space_in_mb	984
num_token	256
endpoint_snitch	Ec2Snitch

図 A.4: altros nosql benchmark cassandra settings

A.2 ワークロード別詳細

ワークロード A: 更新処理

- 読み取り：50%
- 更新：50%
- 4 ノード クラスタ: 5 千万レコード (1 レコード 1KB)
- 10 ノード クラスタ: 1 億レコード
- 20 ノード クラスタ: 2 億レコード

クエリ

付録 A NoSQL 性能評価：MongoDB, Cassandra との比較 ワークロード別詳細

Table 4.1 Evaluated queries for Workload A

Couchbase N1QL	MongoDB Query	Cassandra CQL
<code>bucket.get(docId, RawJsonDocument.class)</code>	<code>db.ycsb.find({'_id': \$1})</code>	<code>SELECT * FROM table WHERE id = \$1 LIMIT 1</code>

図 A.5: altros nosql benchmark workload A query

結果



図 A.6: altros nosql benchmark workload A result

ワークロード E: レンジスキャン

- 読み取り：95%
- 更新：5%
- 4 ノード クラスター: 5 千万レコード (1 レコード 1KB)

付録 A NoSQL 性能評価：MongoDB, Cassandra との比較 クロード別詳細

- 10 ノード クラスタ: 1 億レコード
- 20 ノード クラスタ: 2 億 5 千万レコード

本検証における問題

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、この検証に対して行われます。以下、原文から引用します。

Because the scan operation is performed over the primary key in Couchbase, the following primary index has been created

```
CREATE PRIMARY INDEX 'ycsb_primary' ON 'ycsb'  
USING GSI WITH {"nodes": [...]}\}
```

ここには、Couchbase のプライマリーインデックスに対する誤解が見えます。原文で、「the primary key in Couchbase(Couchbase のプライマリーキー)」と呼ばれているのは、実際には、キーバリューストアとしてドキュメントが格納される際のキー（ドキュメント ID）であり、インデックスの概念とは異なるものです。また、Couchbase Server のプライマリーインデックスは、そのバケット（上記の yscb）に対する、あらゆるクエリを可能にする（インデックスサービスがそのバケットに対する全件スキャンを可能にする）ためのものであって、そのキー（ドキュメント ID）を使った検索を最適化するものではありません（上記の DDL では、バケット名 yscb のみが指定されていることに注意）。後出のクエリの実行を最適化するためには、以下のように、実際にクエリで使われている `META().id` を利用するためのインデックスを、（Couchbase Server の概念におけるセカンダリーインデックスとして）作成する必要があります。

付録 A NoSQL 性能評価：MongoDB, Cassandra との比較

```
CREATE INDEX 'yscb_primary' on 'yscb'(META().id) using GSI;
```

■コラム:

Couchbase Server の N1QL クエリは、SQL を JSON へのクエリとして拡張したものです。キーバリューストアのキーとしてのドキュメント ID は、JSON データを構成する要素ではなく、メタデータの位置づけとなるため、上記のような 'META().id' という指定が用いられます。

クエリ

Table 4.2 Evaluated queries

Couchbase N1QL	MongoDB Query	Cassandra CQL
<pre>SELECT RAW meta().id FROM `yscb` WHERE meta().id >= \$1 ORDER BY meta().id LIMIT \$2</pre>	<pre>db.yscb.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.7: altros nosql benchmark workload E query

結果

付録 A NoSQL 性能評価：MongoDB, Cassandra との比較

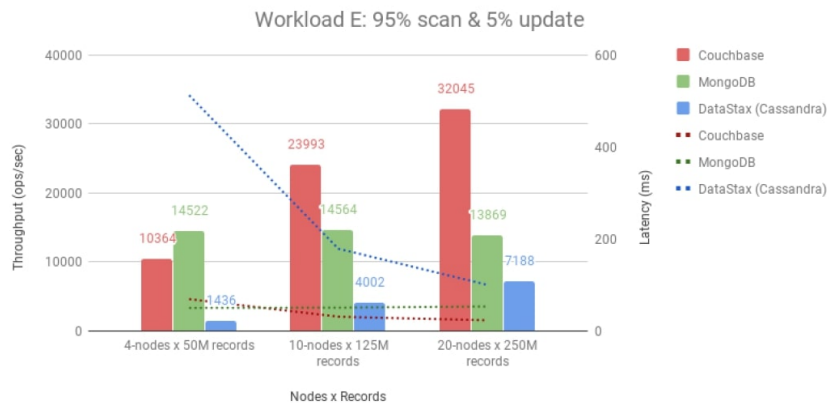


Figure 4.2.3 Performance results under Workload E on 4-, 10-, and 20-node clusters

図 A.8: altros nosql benchmark workload E result

クエリ 1：ページネーション（OFFSET と LIMIT によるフィルター）

- 読み取り： 100%
- 4 ノード クラスター: 5 百万顧客レコード (1 レコード 4KB)
- 10 ノード クラスター: 2 千 5 百万顧客レコード
- 20 ノード クラスター: 5 千万顧客レコード

クエリ

付録 A NoSQL 性能評価：MongoDB, Cassandra, Redis の比較

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.9: altros nosql benchmark query 1 query

結果

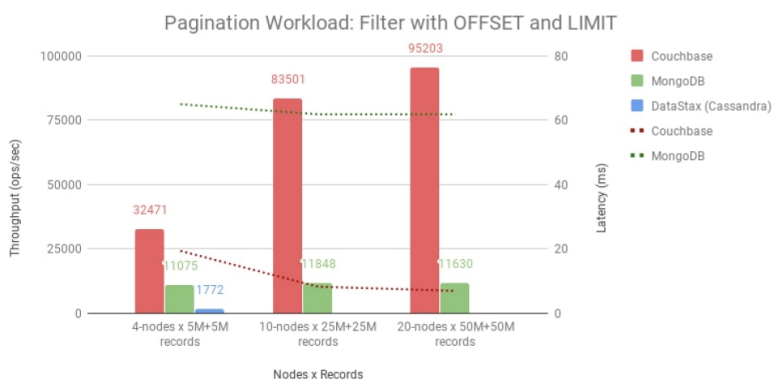


Figure 4.3.3 Performance results under the Pagination Workload on 4-, 10-, and 20-node clusters

図 A.10: altros nosql benchmark query 1 result

クエリ 2：JOIN（テーブル結合）

- 読み取り： 100%
- 4 ノード クラスター: 5 百万顧客レコード、5 百万受注レコード (1 レコード 4.5KB)

付録 A NoSQL 性能評価：MongoDB, Cassandra との比較

- 10 ノード クラスタ: 2 千 5 百万顧客レコード、2 千 5 百万受注レコード
- 20 ノード クラスタ: 5 千万顧客レコード、5 千万受注レコード

クエリ

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>\$r1 = db.customer.find({ address.zip: \$1 }, { address.zip: 1, order_list: 1 }) \$r2 = db.order.aggregate([{ \$match: { \$and: [{ _id: { \$in: \$r1.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.11: altros nosql benchmark query 2 query

結果

付録 A NoSQL 性能評価：MongoDB, Cassandra, Couchbase の比較

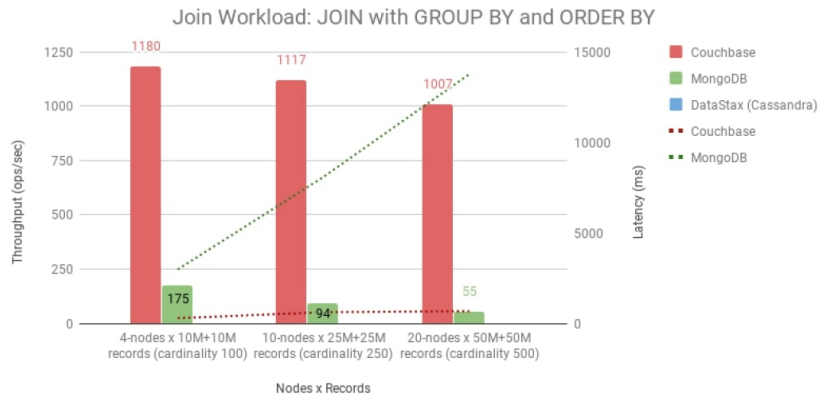


Figure 4.4.3.1 Performance results under Join Workload on 4-, 10-, and 20-node clusters

図 A.12: altros nosql benchmark query 2 result 1 of 2

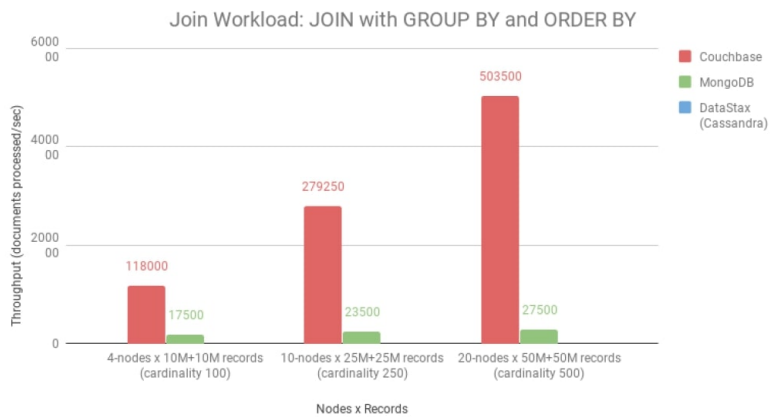
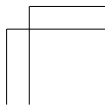
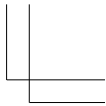
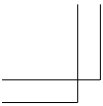
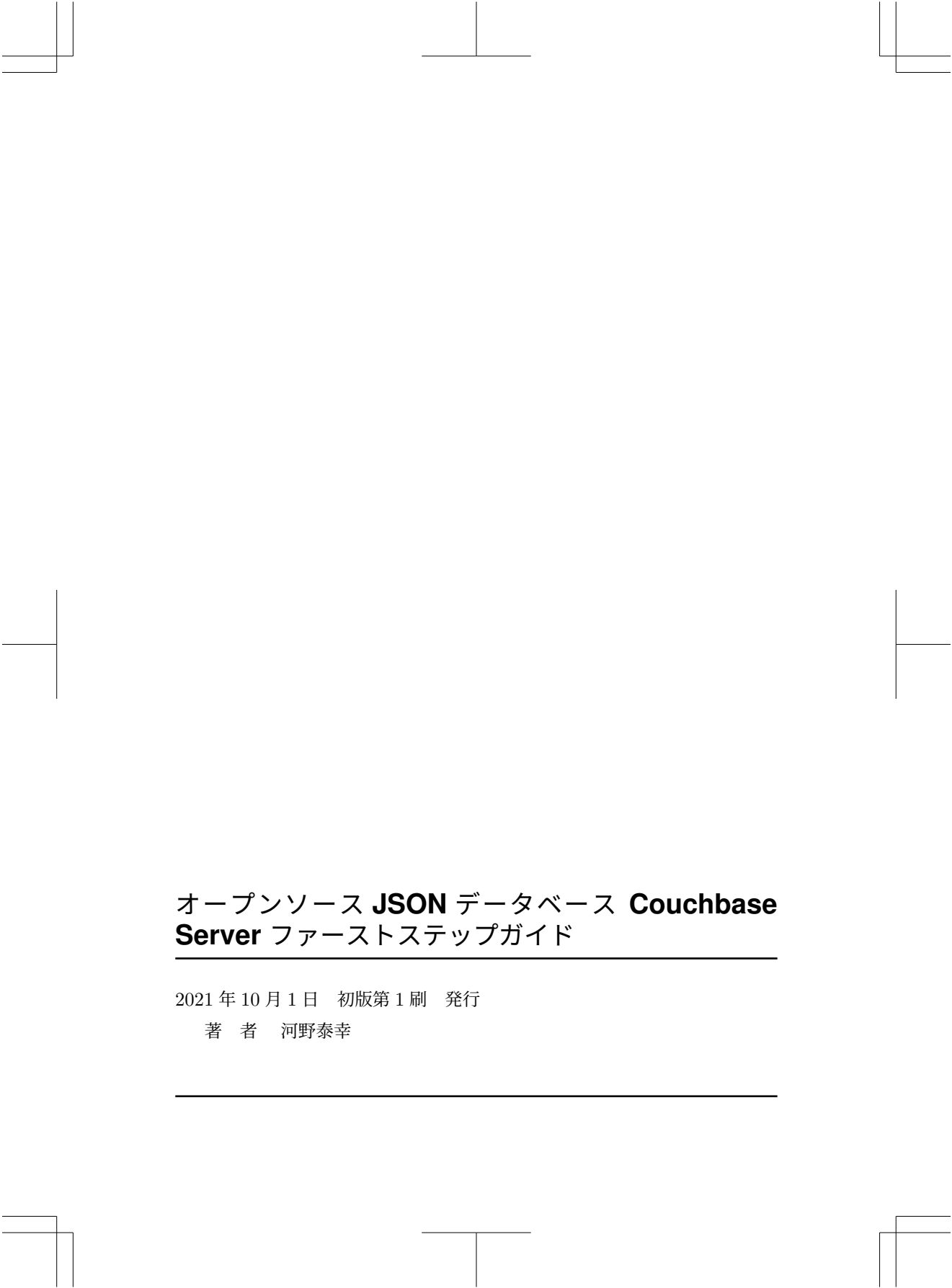


Figure 4.4.3.2 Number of documents processed under Join Workload on 4-, 10-, and 20-node clusters

図 A.13: altros nosql benchmark query 2 result 2 of 2





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

2021 年 10 月 1 日 初版第 1 刷 発行

著 者 河野泰幸
