Iceberg REST Lab

Apache Polaris + PyIceberg で Iceberg REST Catalog を動かして確かめた記録

dobachi

2026-07-21

目次

1 Iceberg REST Lab

> ⚠ この文書の位置づけ
>
> **これは著者による個人的な実験記録であり、本番環境での利用を想定して書かれたものではありません。**
>
> - 特定時点(2026 年 7 月)のスナップショットであり、**内容は急速に陳腐化します**
> - ローカルの単一ノード環境での確認であり、**性能やスケールについては何も言えません**
> - **本番システムの設計・構築の根拠として、そのまま用いないでください。**判断に使う数値・バージョン・仕様は、必ず引用元の一次情報で確認してください
> - 記述の誤りによって生じた結果について、著者は責任を負いません
>
> 技術的な議論のたたき台や、自分で試すときの出発点として使っていただくことを想定しています。

Apache Polaris と PyIceberg を使い、Iceberg REST Catalog をローカルで動かして**文献で読んだことを実際に確かめた**記録です。

Version: 2026-07-21 / 検証基準日: 2026-07-17 / 対象: Apache Iceberg 1.11.0 / PyIceberg 0.11.1 / Apache Polaris 1.6.0

> Version はこのページを更新した日付、検証基準日は実際に動かして確認した時点です。Iceberg 周辺は変化が速いため、**検証基準日から離れるほど内容は古くなります**。

姉妹リポジトリの**調査報告書**が文献調査、こちらがその実地検証にあたります。

1.1 何をした記録か

Iceberg の情報は「仕様がそう定めている」「ある実装がそう振る舞う」「ブログにそう書いてあった」が混同されたまま流通しがちです。この 3 つは一致しません。

そこで、**仕様の記述を引用し、対応する挙動を実際に走らせ、観測結果を並べる**という形で確かめました。詳しくは 00. 実験のコンセプト を参照してください。

`docker compose down -v` でまっさらにした状態から、**サンプル 8 本すべての完走を確認しています**(Docker 29.6.1 / Compose v5.3.0 / Python 3.10 / WSL2)。

1.2 目次

1.2.1 実験の前提

#	ドキュメント	内容
00	実験のコンセプト	何を検証し、何を検証していないか
01	環境構成と起動処理	4 コンテナの構成、ブートストラップが実際にやっていること

1.2.2 実験

#	ドキュメント	対応するサンプル
10	接続とケイパビリティ確認	samples/00_connect.py
11	基本的な CRUD と既定値	samples/01_basic_crud.py
12	スキーマ進化と field ID	samples/02_schema_evolution.py
13	パーティション進化と hidden partitioning	samples/03_partition_evolution.py
14	タイムトラベルと branch / tag	samples/04_time_travel.py
15	メタデータ三層構造を覗く	samples/05_metadata.py
16	REST API を直接叩く	samples/06_rest_api_raw.py
17	PyIceberg の制約を実証する	samples/07_pitfalls.py

1.2.3 まとめ

#	ドキュメント	内容
90	分かったこと	実験を通して確認できたことの集約

1.3 自分で動かす

```
git clone https://github.com/dobachi/iceberg-rest-lab.git
cd iceberg-rest-lab

# 1. 環境を起動(初回はイメージ取得で数分かかります)
make up
```

```
# 2. 表示された CLIENT_ID / CLIENT_SECRET を .env に書く
cp .env.example .env
$EDITOR .env

# 3. Python 依存を入れる
python -m venv venv && source venv/bin/activate
pip install -r requirements.txt

# 4. サンプルを動かす
make sample
```

必要なもの: Docker（compose v2）、Python 3.10 以上。

手順の詳細と、つまずきやすい点は 01. 環境構成 にまとめています。

1.4 ライセンス

対象	ライセンス
コード（compose.yaml / scripts/ / samples/ / Makefile）	Apache License 2.0
ドキュメント（docs/ / README.md）	CC BY 4.0

compose.yaml と scripts/bootstrap.sh は Apache Polaris（Apache-2.0）公式 guides の派生物です。加えた変更は NOTICE に記載しています。

ドキュメント中の引用部分の権利は、それぞれの権利者に帰属します。

2 00. 実験のコンセプト

このラボは、Apache Iceberg と Iceberg REST Catalog について**文献で読んだことを、実際に動かして確かめる**ために作りました。姉妹リポジトリの調査報告書で述べている主張のうち、手元で検証できるものをここで検証しています。

2.1 なぜ動かして確かめるのか

Iceberg の情報は、次の 3 つが混同されたまま流通しがちです。

- **仕様がそう定めている**こと
- **ある実装がそう振る舞う**こと
- **ブログにそう書いてあった**こと

この 3 つは一致しません。たとえばテーブル仕様 v3 は deletion vector を定義していますが、PyIceberg 0.11.1 は v3 を書けません。仕様を読んだだけでは「v3 が使える」と誤解します。逆に、実装を触っただけでは「これは仕様上の制約なのか、この実装の都合なのか」が区別できません。

そこで本ラボでは、**仕様の記述を引用し、対応する挙動を実際に走らせ、観測結果を並べる**という形をとっています。一致すれば裏が取れたことになり、食い違えば、それ自体が記録すべき事実になります。

2.2 何を検証対象にしたか

検証したのは、次の観点です。

- **既定値** --- 新規テーブルの format-version、削除モードなど。「既定でどうなるか」は設計判断に直結しますが、文献では曖昧なまま語られがちです。
- **メタデータ構造** --- 三層構造を実際に開いて、どこに何が入っているかを見ます。
- **進化の仕組み** --- スキーマ進化とパーティション進化が、既存データを書き換えずに成立する仕組みを確認します。
- **REST プロトコル** --- コミットがどういう HTTP のやりとりで実現されているか。特に楽観的並行制御の実際の挙動。
- **実装の制約** --- PyIceberg で何ができないか。これは「できること」より重要な場合があります。

2.3 検証しなかったこと

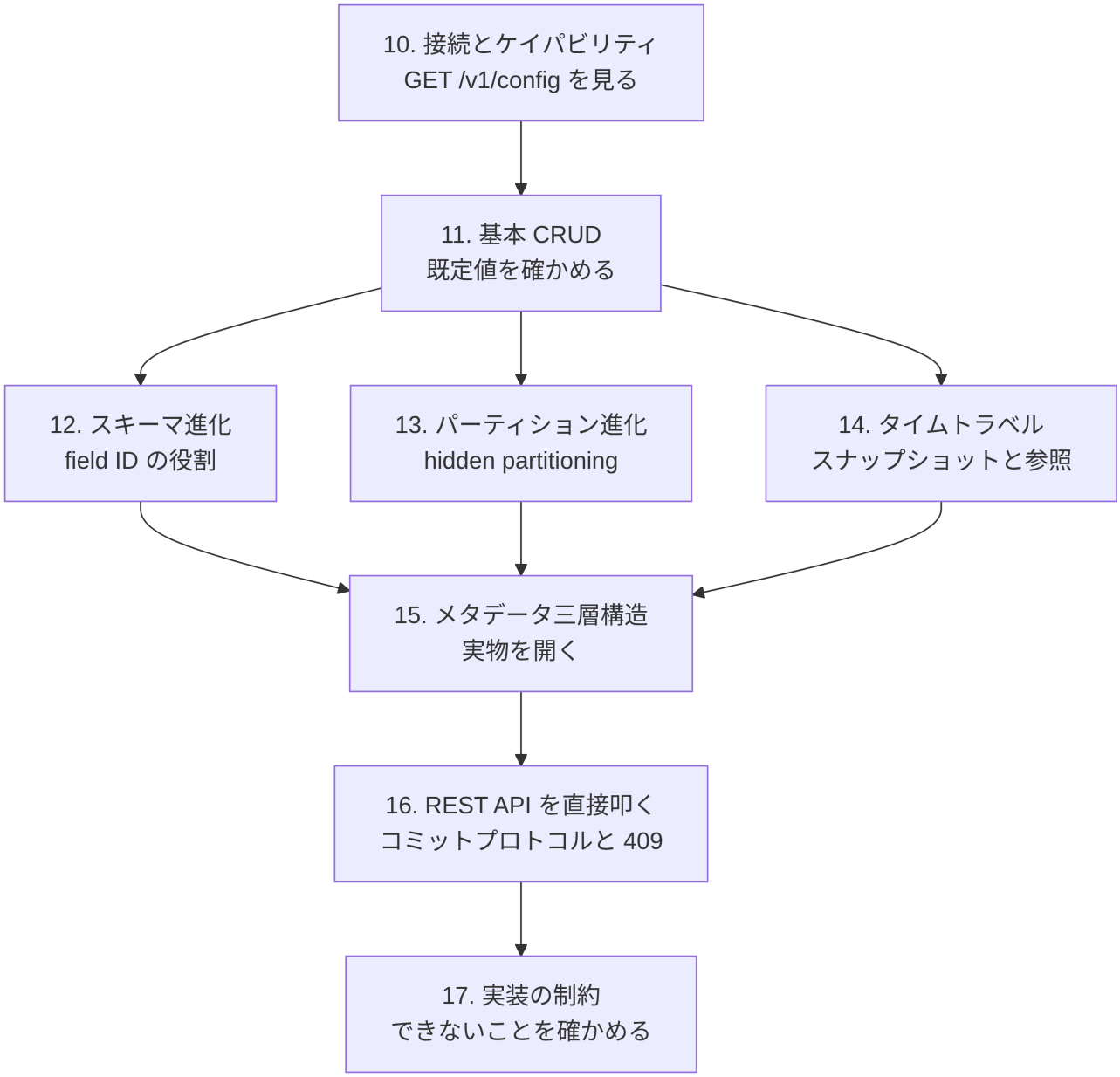
範囲を明示しておきます。以下は本ラボでは扱っていません。

対象	理由
性能・スケール	ローカルの単一ノード環境では、意味のある測定ができません
Spark / Flink / Trino との連携	環境が大きくなりすぎるため。compaction や MERGE INTO はここでは試せません
本番相当の認証（OIDC / 外部 IdP）	学習用に `/v1/oauth/tokens` を使っています。これは仕様上、削除予定のエンドポイントです
永続化	Polaris のメタデータは in-memory です。`docker compose down` で消えます
マルチテナント・権限設計	principal とロールは動かすための最小構成のみです

つまり本ラボは、**仕様と実装の挙動を確かめる**ためのものであり、本番構成の雛形ではありません。

2.4 実験の構成

サンプルは、下の層から順に理解できるように並べています。



前半(10〜13)は Iceberg のテーブルとしての振る舞い、中盤(14〜15)はそれを支えるメタデータ構造、後半(16〜17)はカタログのプロトコルと実装の限界、という順です。

2.5 検証環境

項目	値
検証基準日	2026-07-17
Apache Iceberg	1.11.0
PyIceberg	0.11.1
Apache Polaris	1.6.0

項目	値
ストレージ	RustFS 1.0.0-beta.8（S3 互換）
実行環境	Docker 29.6.1 / Compose v5.3.0 / Python 3.10 / WSL2

構成の詳細と、なぜこの組み合わせを選んだかは 01. 環境構成 に書いています。

2.6 結果の読み方

各サンプルのドキュメントは、次の順で書いています。

1. **確かめたいこと** --- 何を検証する実験か
2. **背景** --- 前提となる概念と仕様の記述
3. **処理の流れ** --- コードが実際に何をしているか
4. **実測結果** --- 実行して観測されたもの
5. **ここから分かること** --- 結論

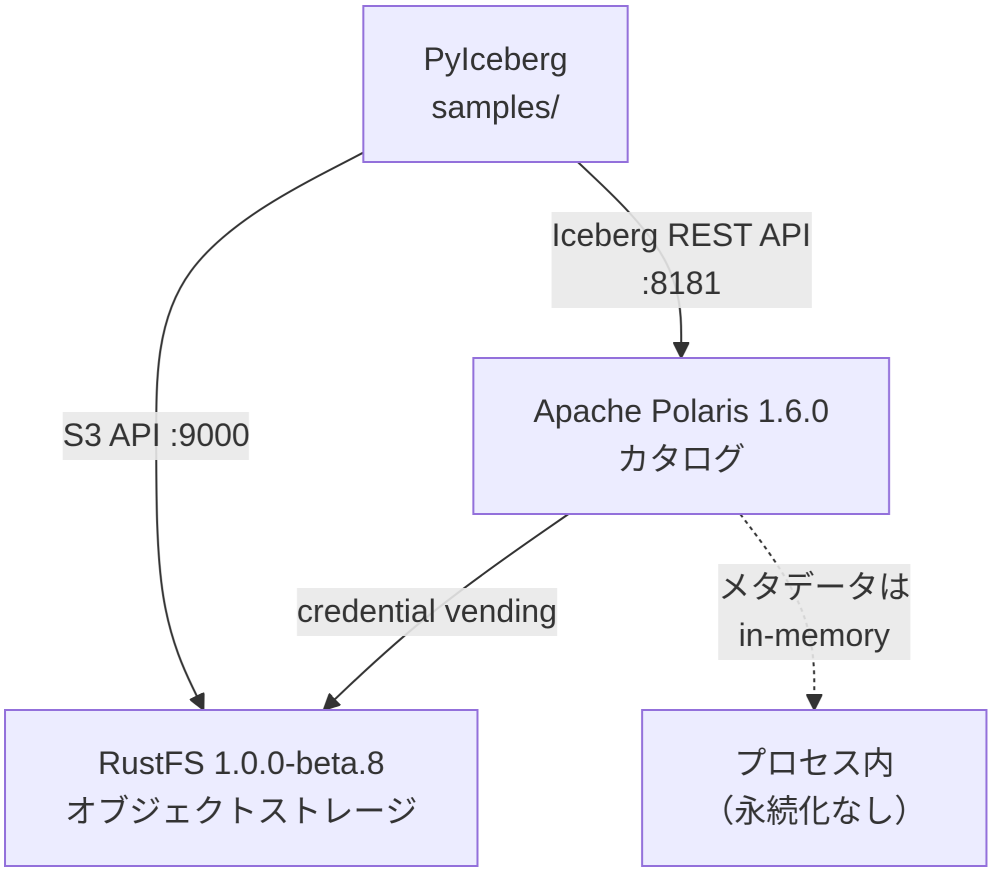
「実測結果」に載せている出力は、検証環境で実際に実行したものをそのまま転記しています。スナップショット ID のように実行ごとに変わる値も、実物を載せています。

実験を通して分かったことは 90. 分かったことのまとめ に集約しています。

3 01. 環境構成と起動処理

このラボは 4 つのコンテナで構成されています。`docker compose up` を実行したとき何が起きるか、なぜその構成なのかを説明します。

3.1 全体構成



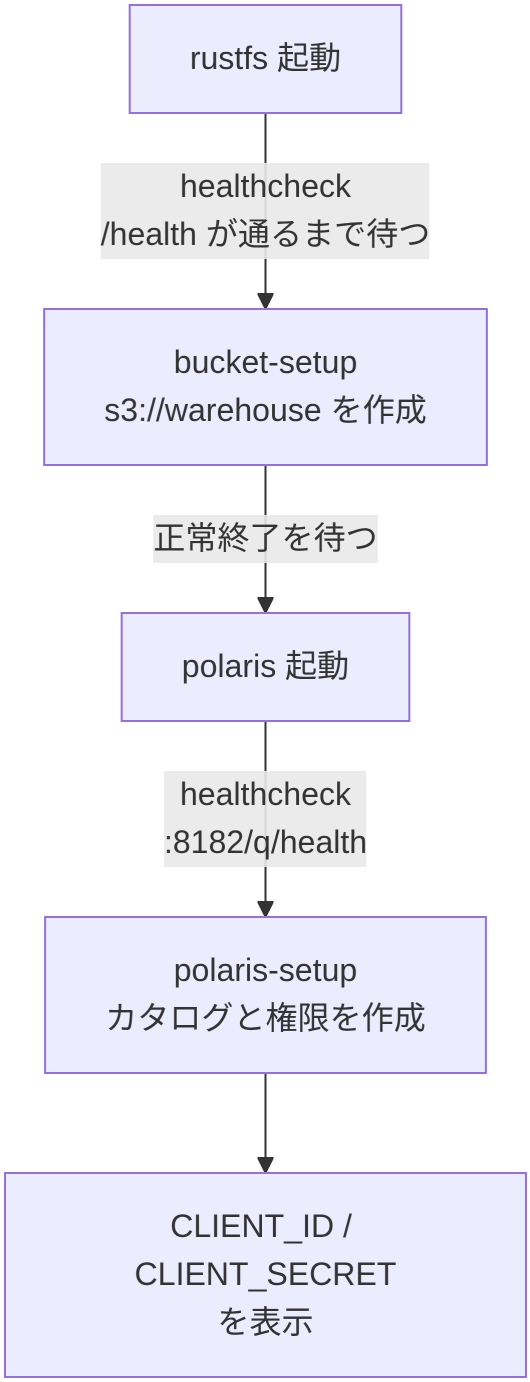
サービス	イメージ	役割	常駐
`rustfs`	`rustfs/rustfs:1.0.0-beta.8`	S3 互換ストレージ。データ本体と metadata JSON を置く	○
`polaris`	`apache/polaris:1.6.0`	Iceberg REST カタログ。:8181 がカタログ API、:8182 が管理・health	○

サービス	イメージ	役割	常駐
bucket-setup	amazon/aws-cli:2.35.21	バケットを作って終了	×
polaris-setup	alpine/curl:8.21.0	カタログ・principal・ロール・権限を作って終了	×

後ろの 2 つは使い捨てで、役目を終えると終了します。`docker compose ps` に残らないのは正常です。

3.2 起動の順序

依存関係は compose の `depends_on` で直列化しています。前段が健全になるまで次は起動しません。



この直列化は必要です。Polaris はカタログ作成時に指定されたバケットへ到達できることを前提にしており、バケットが無い状態で進めると、後段のコミットで初めて失敗します。

3.3 ブートストラップが実際にやっていること

`scripts/bootstrap.sh` は Polaris の**管理 API** を curl で叩きます。ここは Iceberg REST Catalog 仕様ではありません。仕様はカタログの「作成」を定義していないため、実装依存の領域です。

処理は 4 段階です。

3.3.1 1. root のトークンを取得

```
curl "${POLARIS_URL}/api/catalog/v1/oauth/tokens" \
  --user "${ROOT_CLIENT_ID}:${ROOT_CLIENT_SECRET}" \
  -H "Polaris-Realm: ${REALM}" \
  -d grant_type=client_credentials \
  -d scope=PRINCIPAL_ROLE:ALL
```

3.3.1.1 scope=PRINCIPAL_ROLE:ALL は何を指定しているか

OAuth2 の scope は本来「このトークンで何をするつもりか」をクライアントが申告するものです。Polaris はこれを **RBAC の「どの principal role を有効化するか」**の指定として使っています。

Polaris の権限は principal に直接付かず、principal role を経由します(後述)。principal が複数の役割を持つとき、トークンごとに使う役割を選べる設計になっており、PRINCIPAL_ROLE:ALL は「割り当てられている役割をすべて有効化する」という意味です。個別に指定する場合は PRINCIPAL_ROLE:<ロール名> と書きます。

有効化された役割は、権限エラーのメッセージにそのまま現れます。

```
Principal 'root' with activated PrincipalRoles '[service_admin]'
and activated grants via '[service_admin, catalog_admin]'
is not authorized for op CREATE_TABLE_DIRECT_WITH_WRITE_DELEGATION
```

「必須」と書いたのは、**この形式でないとトークン自体が取れない**ためです。実際に試すと次のようになりました。

指定	結果
PRINCIPAL_ROLE:ALL	HTTP 200、トークン取得
PRINCIPAL_ROLE:lab_principal_role	HTTP 200、トークン取得
catalog（PyIceberg の既定値）	**HTTP 400 invalid_scope**
指定なし	**HTTP 400 invalid_scope**

PyIceberg は CATALOG_SCOPE = "catalog" を既定値として送るため、scope を明示しないと Polaris ではトークン取得の時点で失敗します。認証や権限の問題ではなくスコープの書式の問題なので、返るのは 401 / 403 ではなく 400 です。

3.3.1.2 ロール名はどこまで効くか

存在しないロール名を指定してもトークンが発行されたため、実装を確認しました。コンテナ内の polaris-runtime-service-1.6.0.jar を逆アセンブルして読んだ結果です。

トークン発行時（TokenRequestValidator）は書式しか見ていません。スコープを空白で分割し、各要素が PRINCIPAL_ROLE: で始まること、接頭辞を除いた残りが空でないことだけを検証します。条件を満たさなければ invalid_scope を返します。**ロールが実在するかどうかは確認していません。**だから PRINCIPAL_ROLE:does_not_exist でもトークンが出ます。

認証時（DefaultAuthenticator）はロール名を実際に使っています。PRINCIPAL_ROLE:ALL なら全ロールを有効化し、そうでなければ principal に付与されたロールを要求された名前で絞り込みます。要求した名前が見つからない場合は警告を記録するだけで、例外は投げません。

実際にログに現れます。

```
WARN [org.apa.pol.ser.aut.DefaultAuthenticator]
  principal=lab_user credentials=InternalPolarisToken{... scope=PRINCIPAL_ROLE:does_not_exist}
  roles=[] Some principal roles were not found in the principal's grants
```

roles=[] のとおり、有効化されたロールは空になっています。ロール名は確かに解釈されており、絞り込みも働いています。

ところが、その状態でもテーブルを作成できました。

スコープ	有効ロール	listNamespaces	createTable（新規名）
PRINCIPAL_ROLE:ALL	全ロール	HTTP 200	HTTP 200
PRINCIPAL_ROLE:does_not_exist	{}（空）	HTTP 200	**HTTP 200**

有効ロールが空のトークンで新規テーブルが作られ、metadata-location が返りました。一方、purgeRequested=true を付けたテーブル削除は同じトークンで **HTTP 403** になったので、認可がまったく効いていないわけではありません。

つまり **1.6.0 のこの構成では、スコープでロールを絞っても実効的な権限は絞られませんでした。**なぜ空のロール集合で書き込みが通るのかまでは追えていません（未確認）。スコープを権限の絞り込み手段として当てにしない前提で扱うのが安全です。

ただし**ここで確認したのは、仕様上 DEPRECATED for REMOVAL とされている /v1/oauth/tokens 経由の挙動**である点に注意してください（このエンドポイントの位置づけは前述のとおりです）。推奨される構成、つまり外部 IdP を使い oauth2-server-uri を設定した場合の挙動は確認していません。認証の経路が変われば、ロールの解決も変わる可能性があります。

なお /v1/oauth/tokens は Iceberg 仕様上 **DEPRECATED for REMOVAL** です（Java 1.6.0 で非推奨、2.0 で削除予定）。ここでは学習の便宜上使っています。

3.3.2 2. カタログを作成

```
{
  "storageConfigInfo": {
    "storageType": "S3",
    "allowedLocations": ["s3://warehouse/lab_catalog"],
    "endpoint": "http://localhost:9000",
    "endpointInternal": "http://rustfs:9000",
    "pathStyleAccess": true,
    "region": "us-east-1"
  }
}
```

ここには 2 つの要点があります。

エンドポイントを内外で分ける。`endpoint` はクライアント（ホスト）から見た URL、`endpointInternal` は Polaris コンテナから見た URL です。取り違えると `Name or service not known` になります。

`roleArn` を書かない。`roleArn` があると Polaris は credential vending のために STS AssumeRole を呼びます。RustFS も MinIO も STS に対応していないため、コミット時に `CommitStateUnknownException: StsException (Status Code: 400)` で落ちます。省略すると Polaris は環境変数の静的な資格情報をそのまま使い、credential vending も機能します。

このエラーは紛らわしい点があります。`CommitStateUnknownException` は仕様上「コミットが成功したか失敗したか分からない」という重大な状態を指しますが、この場合の実態は単なる設定ミスです。

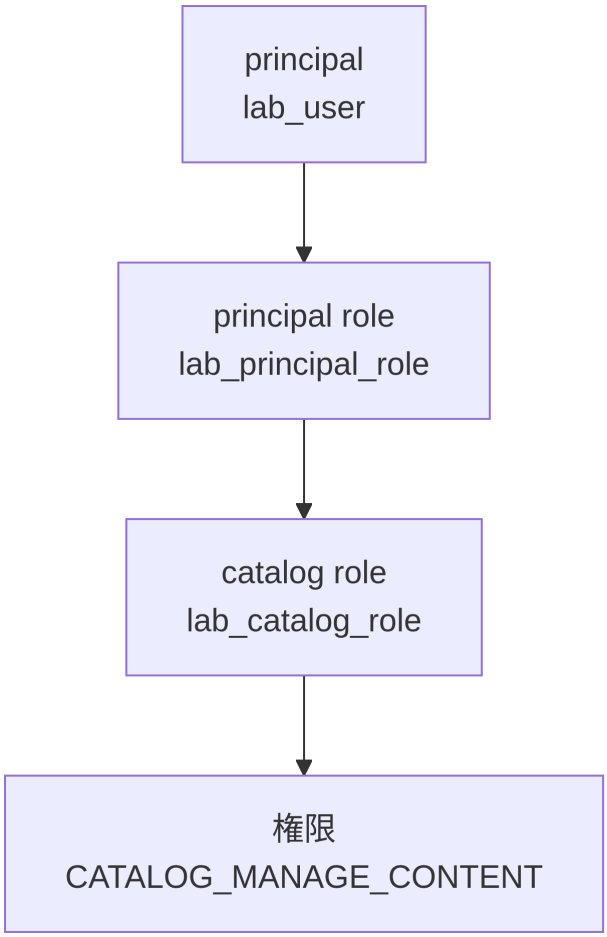
3.3.3 3. principal を作成

`lab_user` という principal を作り、`clientId` / `clientSecret` を受け取ります。この資格情報は**作成時に一度だけ**払い出されます。

principal が既に存在する場合（`docker compose down` だけして volume を残した場合など）は作成に失敗するため、root の資格情報にフォールバックします。作り直すには `docker compose down -v` が必要です。

3.3.4 4. ロールと権限を紐付ける

Polaris の RBAC は 2 層です。



principal に直接権限は付きません。principal role を経由して catalog role に到達し、そこに権限が付く構造です。この 2 段構えは、principal を増やしても権限定義を再利用できるようにするためのものです。

3.4 設定はどこから読まれるか

サンプルは samples/_common.py 経由でカタログに接続します。設定の優先順位は次のとおりです。

1. **export 済みの環境変数**（最優先）
2. **リポジトリ直下の .env**
3. **_common.py の既定値**（root / s3cr3t など）

.env は _common.py が起動時に読み込みます。make up が表示する CLIENT_ID / CLIENT_SECRET をここに書きます。書き忘れると既定値の root にフォールバックし、テーブル作成時に次のエラーになります。

```
ForbiddenException: Principal 'root' with activated PrincipalRoles '[service_admin]'
... is not authorized for op CREATE_TABLE_DIRECT_WITH_WRITE_DELEGATION
```

root は管理者ですが、カタログのコンテンツを操作する権限は持っていません。Polaris が管理操作とデータ操作の権限を分けているためです。

3.5 接続時のプロパティ

_common.py がカタログに渡している設定のうち、S3 互換ストレージで問題になりやすいものを挙げます。

キー	値	なぜその値か
uri	http://localhost:8181/api/catalog	末尾に/v1 は**付けない**
warehouse	lab_catalog	S3 パスではなく**カタログ名**
scope	PRINCIPAL_ROLE:ALL	この書式でないとトークンが取れない
s3.endpoint	http://localhost:9000	設定すると自動的に path-style になる
s3.region	us-east-1	値は任意だが、指定しないと遅いか失敗する

それぞれ理由があります。

uri に /v1 を付けない。PyIceberg 側が付与するためです。ソースでは url + "/v1/" として組み立てており（catalog/rest/__init__.py L551）、エンドポイントのテンプレートも API_PREFIX = "/v1/{prefix}" です。こちらで /v1 を書くと /api/catalog/v1/v1/... になり 404 になります。

warehouse にカタログ名を渡す。ここは実装間で意味が割れる箇所です。この値は GET /v1/config のクエリパラメータとしてサーバに送られ、サーバが「どのカタログの設定を返すか」を決めるために使います。仕様はこの値の意味を定めていないため、解釈はサーバ次第です。Hadoop カタログのように S3 パス（s3://bucket/warehouse）を期待する実装もありますが、Polaris は登録済みのカタログ名を期待します。実際に両方を投げると次のようになりました。

warehouse の値	結果
lab_catalog（カタログ名）	HTTP 200、overrides.prefix = lab_catalog が返る
s3://warehouse/lab_catalog（パス）	**HTTP 404 Unable to find warehouse s3://warehouse/lab_catalog**

エラーメッセージが「warehouse が見つからない」と言っている点に、この値をカタログの識別子として扱っていることが表れています。

scope は書式が検証される。前述のとおり、PRINCIPAL_ROLE: 形式でないと HTTP 400 invalid_scope でトークン取得に失敗します。

s3.endpoint を設定すると path-style になる。PyArrow の S3FileSystem は force_virtual_addressing の既定が False で、その意味は「endpoint_override が空のときだけ virtual addressing を使う」です。エンドポイントを指定した時点で条件から外れ、path-style が選ばれます。AWS S3 以外では bucket をホスト名に含める virtual-hosted style が使えないため、この挙動が必要です。

s3.region は値そのものより「指定してあること」が効く。未指定だと PyIceberg は_cached_resolve_s3_region(bucket=
でバケットのリージョンを問い合わせます（io/pyarrow.py L209）。これは AWS S3 に対する照会なので、RustFS や MinIO では失敗するか、タイムアウトまで待たされます。RustFS はリージョンを区別しないため、指定する値自体は何でも構いません。

`s3.path-style-access` は **PyIceberg に存在しません**。Java 版 Iceberg のキーであり、書いてもエラーにならず黙って無視されます。詳しくは 17. 実装の制約 で扱います。

3.6 なぜこの組み合わせか

3.6.1 Apache Polaris

セルフホストで REST カタログを試す用途では、現時点で最有力だと判断しました。

- 2026-02-19 に ASF の Top-Level Project に卒業しています
- 外部 DB 不要で立ち上がり、カタログ・principal・ロール・権限まで自動生成できます
- IRC 仕様のリファレンス実装的な位置づけで、ここで学んだことが他の実装にも通じます
- `guides/` 配下に用途別 compose（`jdbc`, `keycloak`, `trino`, `flink`, `spark`）が揃っており、段階的に拡張できます

対抗馬の評価は報告書の該当章にまとめています。

3.6.2 MinIO ではなく RustFS

MinIO が実質的に使えなくなったためです。

時期	出来事
2025-10-15	セキュリティ修正（権限昇格）。GitHub 上の最終リリース
2025-10-18	その修正のイメージが未公開のまま `working as intended` でクローズ
2026-04-25	`minio/minio` リポジトリがアーカイブ（read-only）

入手可能な最新の公開イメージ（2025-09-07）は、2025-10-15 の権限昇格 CVE 修正を含みません。Apache Polaris 公式も MinIO ガイドに maintenance mode の警告を出し、PR #3482 で RustFS の例を追加しています。

ここで流通している誤解を 1 つ訂正しておきます。「MinIO が Apache-2.0 から AGPLv3 に変更したのが理由」という説明を見かけますが、**MinIO のライセンスは AGPLv3 のまま変わっていません**。変わったのは保守方針と配布形態です。

なお RustFS 自体も **1.0.0 の安定版には未到達**です（2026-07-16 時点で `1.0.0-beta.10-preview.4`）。ラボ用途としては足りますが、本番の判断材料にはしないでください。

3.6.3 イメージタグの固定

Polaris 公式 quickstart は `apache/polaris:latest` と `rustfs:1.0.0-alpha.81` を使っていますが、本ラボではすべて固定しています。

- `apache/polaris:latest` の実体バージョンが分かりません

- 公式の `guides/quickstart` と `guides/rustfs` が別バージョンの RustFS を使っており、公式内でも不整合があります
- `apache/iceberg-rest-fixture` は `latest`（2026-04-29）がバージョン付き最新の `1.10.1`（2025-12-22）より新しく、中身が一致しない可能性があります

再現性のために固定が必要だと判断しました。

3.7 永続化について

Polaris のメタデータは **in-memory** です。`docker compose down` でテーブル定義は消えます（データ本体は volume に残りますが、カタログから参照されなくなるため孤立します）。

永続化する場合は Polaris 公式の `guides/jdbc`（PostgreSQL）を参照してください。

4 10. 接続とケイパビリティ確認

最初のサンプル samples/00_connect.py は、PyIceberg から Apache Polaris の REST カタログに接続し、接続時に何が起きているかを確認します。REST カタログの接続は URI を指定して終わりではなく、GET /v1/config によるサーバ設定の受け取りと、クライアント設定との合成が挟まります。

ここではその合成結果と、サーバが宣言するケイパビリティ（対応エンドポイント一覧）を見ます。

4.1 確かめたいこと

- GET /v1/config が返す defaults / overrides が、クライアント側の設定とどう合成されるか
- 合成後のカタログ設定に、サーバ由来の値がどう現れるか
- サーバが endpoints フィールドで何を宣言しているか
- PyIceberg 側からその宣言を参照できるか

4.2 背景

Iceberg REST Catalog 仕様は、設定エンドポイントについて「All REST clients should first call this route」と定めています。クライアントはカタログに対する最初のリクエストとしてこれを呼び、返ってきた設定を自身の設定と合成してから以降の操作に入ります。

合成の優先順位は次のとおりです。後に来るものが勝ちます。

```
defaults  →  クライアント設定  →  overrides
 ↑弱い                              ↑最強
```

defaults はサーバの推奨値で、クライアントが同じキーを指定していればクライアント側が勝ちます。overrides はサーバが強制する値で、クライアントの指定を上書きします。Polaris の場合、マルチテナントのルーティングに使う prefix などが overrides に入ります。

もうひとつの要素が endpoints です。これはサーバが対応する REST 操作の一覧で、ケイパビリティの宣言にあたります。endpoints を返さないサーバは、仕様が定めるデフォルトセット（13 個）だけをサポートしていると解釈されます。このデフォルトセットにはビュー関連も scan planning も含まれません。PyIceberg 0.11.0 で ConfigResponse の supported endpoints ネゴシエーションが実装され、未対応の操作を呼んだときにエラーが明確になりました。

なお、Polaris への接続には PyIceberg の既定値のままでは通らない設定がいくつかあります。詳細は samples/_common.py のコメントにまとまっていますが、要点は次の 3 点です。

- scope の既定値は catalog だが、Polaris では PRINCIPAL_ROLE:ALL が必要
- warehouse は S3 パスではなくカタログ名を指定する
- s3.region は S3 互換ストレージが相手でも実質必須（未指定だとリージョン解決に失敗するか極端に遅くなる）

4.3 処理の流れ

1. load_catalog() でカタログに接続します。この時点で OAuth2 の client_credentials フローによるトークン取得と GET /v1/config の呼び出しが行われ、結果が catalog.properties にマージ済みで保持されます。
2. マージ後の設定を列挙します。秘密情報はマスクします。

```
for k, v in sorted(catalog.properties.items()):
    if any(s in k.lower() for s in ("secret", "credential", "token", "password")):
        v = "***"
    print(f"    {k} = {v}")
```

3. サーバが宣言したエンドポイントを、PyIceberg の内部属性から取得します。取得できない場合はその旨を表示します。

```
endpoints = getattr(catalog, "_endpoints", None)
```

4. namespace lab を作成(既存なら何もしない)し、一覧を表示します。

4.4 実測結果

接続と設定のマージ結果は次のようになりました。

```
--- カタログに接続します
    接続先: http://localhost:8181/api/catalog
    warehouse: lab_catalog
    型: RestCatalog

--- マージ後のカタログ設定 (GET /v1/config の結果が反映済み)
    credential = ***
    default-base-location = s3://warehouse/lab_catalog
    namespace-separator = %1F
    prefix = lab_catalog
    s3.access-key-id = rustfsadmin
    s3.endpoint = http://localhost:9000
    s3.region = us-east-1
    s3.secret-access-key = ***
    scope = PRINCIPAL_ROLE:ALL
    type = rest
    uri = http://localhost:8181/api/catalog
    warehouse = lab_catalog
```

このうち default-base-location はサーバの defaults、namespace-separator と prefix はサーバの overrides に由来します。クライアント側は一切指定していません。生のレスポンスは samples/06_rest_api_raw.py で確認でき、次の内容でした。

```
{
  "defaults": {
    "default-base-location": "s3://warehouse/lab_catalog"
  },
  "overrides": {
    "namespace-separator": "%1F",
    "prefix": "lab_catalog"
  },
  "endpoints": [ ... ]
}
```

endpoints には 36 個のエントリが並びます。仕様のデフォルト 13 個に加えて、ビュー関連（/views、/views/rename、register-view）と、Polaris 独自の generic-tables・policies 系エンドポイントが宣言されていました。前者は Iceberg 仕様の範囲、後者は polaris/v1/... というパスを持つ拡張です。

一方、PyIceberg 側からの参照は取得できませんでした。

```
--- サーバが宣言したエンドポイント（ケイパビリティ）
    (PyIceberg の内部表現から取得できませんでした)
    生のレスポンスは 06_rest_api_raw.py で確認できます。
```

_endpoints という属性名は PyIceberg の内部実装に依存しており、0.11.1 の RestCatalog では同名の属性を参照できませんでした。ネゴシエーション自体が行われていないのか、別の名前で保持されているのかは未確認です。サーバの宣言内容そのものは前述のとおり生のレスポンスで確認できます。

namespace の作成は成功しました。

```
--- namespace を作成/確認します
    namespace 一覧: [('lab',)]
```

4.5 ここから分かること

REST カタログでは、クライアントの設定ファイルに書いた内容がそのまま使われるとは限りません。サーバが overrides で上書きする項目があり、実際に有効な設定は接続後に初めて確定します。設定の食い違いを調べるときは、クライアント側の記述ではなく合成後の値（catalog.properties）を見る必要があります。

endpoints の宣言は、カタログ実装の差異をクライアントが事前に把握するための仕組みです。宣言されていない操作は、実行してから 404 や 405 に遭遇するのではなく、クライアント側で早期に判断できます。ただし PyIceberg 0.11.1 の公開 API からこの一覧を読む手段はこのサンプルでは見つかっておらず、確認したい場合は GET /v1/config を直接叩くのが確実です。

また、Polaris のように仕様の範囲外のエンドポイントを同じ endpoints 配列で宣言する実装があります。パスに polaris/v1/ が含まれるものがそれにあたり、他のカタログ実装に移す際は使えません。ポータビリティを気にする場合は、宣言されているからといって使ってよいとは限らない点に注意が必要です。

次は 11. 基本的な CRUD と既定値 です。

5 11. 基本的な CRUD と既定値

samples/01_basic_crud.py は、テーブルの作成・追加・読み取り・削除・上書きをひととおり実行します。操作そのものは単純ですが、このサンプルの目的は既定値の確認にあります。

Iceberg の新機能（format version 3、deletion vector、merge-on-read）は仕様上は利用可能でも、テーブルを普通に作っただけでは有効になりません。何もしなければどの設定が使われるのかを、実際のテーブルから読み取ります。

5.1 確かめたいこと

- 新規テーブルの format-version の既定値
- write.delete.mode の既定値
- append 1 回でスナップショットがいくつ増えるか
- delete がどの operation として記録されるか
- 条件つき overwrite の結果

5.2 背景

format version は、テーブルが使えるメタデータ機能を決めます。v2 で row-level delete（position delete / equality delete ファイル）が導入され、v3 で deletion vector やデフォルト値つき列などが追加されました。ただしライブラリの既定値は最新版ではありません。PyIceberg では TableProperties.DEFAULT_FORMAT_VERSION が 2 で、明示しない限り v2 のテーブルができます。

write.delete.mode は、行削除をどう実装するかを決めるテーブルプロパティです。copy-on-write は削除対象を含むデータファイルを丸ごと書き直します。merge-on-read は削除マーカーだけを書き、読み取り時に適用します。既定は copy-on-write で、format version を v3 に上げても deletion vector が自動的に使われるわけではありません。PyIceberg の書き込みパスは merge-on-read を指定しても警告を出して copy-on-write に落ちます（samples/07_pitfalls.py で確認できます）。

もう 1 点、コミットとスナップショットの関係があります。Iceberg のコミットは、新しいメタデータ JSON を書き、カタログ側でポインタを原子的に差し替えることで成立します。1 回のコミットが 1 個のスナップショットと 1 個のメタデータ JSON を生むのは、実装の都合ではなく原子性を得るための構造です。高頻度コミットがメタデータの肥大を招くのはこのためです。

5.3 処理の流れ

1. lab.trips を作り直します。スキーマは trip_id（required）、city（required）、fare、ts（required）の 4 列です。

2. 作成直後のテーブルから既定値を読みます。

```
fmt = tbl.metadata.format_version
assert fmt == 2, f"期待は 2 でしたが {fmt} でした"
delete_mode = tbl.properties.get("write.delete.mode", "copy-on-write (未設定時の既定)")
```

3. 5 行を append し、スナップショット数と summary を表示します。ここで dict(snap.summary) は使えません。Summary は Mapping を継承していますが __iter__ がキーではなくタプルを返すため、dict() が AttributeError: 'tuple' object has no attribute 'lower' で落ちます。公開プロパティの additional_properties を使います。

```
for k, v in sorted(snap.summary.additional_properties.items()):
    print(f"      {k} = {v}")
```

4. 全件読み取りと、述語プッシュダウンつきの読み取りを行います。

```
df = tbl.scan(row_filter="city == 'tokyo'", selected_fields=("trip_id", "fare")).to_arrow()
```

5. もう 1 回 append し、スナップショットが増えることを確認します。
6. delete_filter="city == 'kyoto'" で削除し、operation を確認します。
7. overwrite_filter="city == 'osaka'" つきの overwrite で、条件に一致する行を置き換えます。

5.4 実測結果

作成直後の既定値は次のとおりでした。

```
--- 新規テーブルの既定値を確認します
    format-version = 2
    → 既定は v2 です。v3 ではありません。
       (PyIceberg の TableProperties.DEFAULT_FORMAT_VERSION = 2)
    write.delete.mode = copy-on-write (未設定時の既定)
```

write.delete.mode はテーブルプロパティとして設定されておらず、get() のフォールバック文字列が表示されています。つまり明示的な値は書かれておらず、読み手側が仕様の既定である copy-on-write として解釈します。

5 行を append した結果です。

```
--- データを append します
    スナップショット数: 1
    現在のスナップショット: id=6414712164395274846
    operation = Operation.APPEND
      added-data-files = 1
      added-files-size = 1780
      added-records = 5
      total-data-files = 1
```

```
      total-delete-files = 0
      total-equality-deletes = 0
      total-files-size = 1780
      total-position-deletes = 0
      total-records = 5
```

summary には追加分(`added-*`)と累計(`total-*`)の両方が入ります。delete ファイル系のカウンタはすべて 0 で、この時点では row-level delete が存在しません。

述語プッシュダウンつきの読み取りでは、`city == 'tokyo'` の 2 行のみ、指定した 2 列だけが返りました。

```
 trip_id   fare
       1 1200.0
       3 2300.0
```

`row_filter` は列統計を使ったファイルプルーニングに使われます。このサンプルではデータファイルが 1 つしかないため枝刈りの効果は見えません。

2 回目の append 後は次のとおりです。

```
--- もう 1 回 append して、スナップショットが増えることを確認します
    スナップショット数: 2
    行数: 6
```

1 行の追加でスナップショットが 1 個増えました。

削除の結果です。

```
--- delete (copy-on-write で実行されます)
    行数: 5
    operation = Operation.OVERWRITE
```

`kyoto` の 1 行が消えて 6 行から 5 行になりましたが、記録された operation は `DELETE` ではなく `OVERWRITE` です。copy-on-write では削除対象を含むデータファイルを、その行を除いた内容で書き直します。ファイルの置き換えが起きているので、意味的には上書きにあたります。summary の operation が `DELETE` になるのは、データファイルを丸ごと削除できる場合(メタデータのみで完結する削除)です。

条件つき overwrite の結果です。

```
 trip_id   city   fare                  ts
       1  tokyo 1200.0 2026-07-01 01:00:00
       2  osaka 8888.0 2026-07-01 02:00:00
       3  tokyo 2300.0 2026-07-01 03:00:00
       6 nagoya 1100.0 2026-07-03 09:00:00
```

`city == 'osaka'` に一致していた 2 行(trip_id 2 と 5)が、渡した 1 行(trip_id 2、fare 8888.0)に置き換わりました。条件に一致する行はすべて消え、渡したデータだけが残ります。UPSERT ではない点に注意が必要です。

5.5 ここから分かること

format version と delete mode の既定は、いずれも保守的な側に倒れています。v3 の機能や merge-on-read を使いたい場合は明示的な指定が要ります。「Iceberg は deletion vector に対応した」という記述と、手元のテーブルが実際に使っている構成は別の話です。既定のままなら v2 + copy-on-write です。

copy-on-write の削除は、対象行が少なくてもファイル単位の書き直しを伴います。1 行消すために数百 MB のファイルを書き直すことが起こり得るので、削除の頻度が高いワークロードでは書き込み量を見積もる必要があります。

コミット 1 回につきスナップショット 1 個とメタデータ JSON 1 個が増えるため、小さな書き込みを繰り返すとメタデータが線形に増えます。スナップショットの整理（expire）とデータファイルの整理は、運用に入る前に方針を決めておく対象です。スナップショットの寿命については 14. タイムトラベルと branch / tag で扱います。

前は 10. 接続とケイパビリティ確認、次は 12. スキーマ進化と field ID です。

6 12. スキーマ進化と field ID

samples/02_schema_evolution.py は、列の追加・改名・型変更・削除を順に実行し、そのたびにスキーマの中身を表示します。注目するのは列名ではなく field ID です。

Iceberg のスキーマ進化がデータファイルの書き直しなしに成立するのは、列の同一性を名前ではなく field ID で管理しているからです。ここではその不変性を実際に確認します。

6.1 確かめたいこと

- 列を追加したとき、既存行の値がどうなるか
- 列を rename したとき field ID が変わるか
- 許可される型プロモーションと、許可されない変更の挙動
- 列を削除したときデータファイルがどうなるか
- 過去のスキーマがテーブルに残るか

6.2 背景

Iceberg 仕様は、データファイルの列の解決方法について次のように定めています（Table Spec, Schemas and Data Types の項）。

> Columns in Iceberg data files are selected by field id. The table schema's column names and order may change after a data file is written, and projection must be done using field ids.

改名については同じく次のとおりです。

> Renaming an existing field must change the name, but not the field ID.

つまり Parquet ファイル側は field ID で列を持ち、読み手はテーブルスキーマの field ID を使って射影します。名前はスキーマ側のラベルにすぎません。これが、rename でデータファイルを書き換えなくてよい理由です。

追加した列を既存ファイルが持っていない場合、読み手はその field ID を見つけられず null を返します。列の削除も、スキーマから field ID を外すだけで、データファイル側には手を触れません。

型プロモーションは、既存のバイト列を再解釈できる方向にだけ許されます。int から long への拡大は v1/v2/v3 すべてで許可されますが、逆方向は値が失われる可能性があるため拒否されます。

なお PyIceberg の `update_schema()` は Iceberg 型を取ります。`pa.string()` のような PyArrow 型を渡すと `NotImplementedError: Cannot visit non-type` になります。

6.3 処理の流れ

1. `lab.evolve` を `id`（long, required）と `name`（string, optional）で作成し、2 行入れます。
2. `add_column` で `email` を追加し、既存行を読み直します。

```
with tbl.update_schema() as us:
    us.add_column("email", StringType(), doc="メールアドレス")
```

3. `rename_column` で `name` を `full_name` に変え、rename 前後の field ID を突き合わせます。

```
before = {f.name: f.field_id for f in tbl.schema().fields}
with tbl.update_schema() as us:
    us.rename_column("name", "full_name")
after = {f.name: f.field_id for f in tbl.schema().fields}
assert before["name"] == after["full_name"], "field ID が変わってしまいました"
```

4. `IntegerType` の `score` 列を追加してから `LongType` に変更し、続けて `IntegerType` に戻そうとします。後者は失敗するはずなので例外を捕まえて表示します。
5. `delete_column` で `score` を削除します。
6. `tbl.metadata.schemas` を列挙し、保持されているスキーマの数と内容を表示します。

6.4 実測結果

初期スキーマと列追加後のスキーマです。

```
--- 初期スキーマ
    [initial]
      field-id=1   id          long       required
      field-id=2   name        string     optional

--- 列を追加します (add_column)
    [after add_column]
      field-id=1   id          long       required
      field-id=2   name        string     optional
      field-id=3   email       string     optional
```

追加された `email` には新しい field ID 3 が割り当てられ、既存の 1 と 2 は動きません。既存 2 行を読むと `email` は null になりました。

```
 id  name email
  1 alice  None
  2   bob  None
```

データファイルは追加時のまま(id と name の 2 列)で、読み手が field ID 3 を見つけられず null を埋めています。

rename の結果です。

```
    [after rename_column]
      field-id=1   id           long       required
      field-id=2   full_name    string     optional
      field-id=3   email        string     optional

    rename 前の 'name' の field-id : 2
    rename 後の 'full_name' の field-id: 2
```

field ID は 2 のまま変わっていません。rename 後も既存 2 行はそのまま読めました。

型プロモーションは int → long が成功しました。

```
    [after int -> long promotion]
      field-id=1   id           long       required
      field-id=2   full_name    string     optional
      field-id=3   email        string     optional
      field-id=4   score        long       optional
```

逆方向は拒否されました。

```
--- 許可されていない型プロモーションを試します (long -> int)
    期待どおり拒否されました: ValidationError
    Cannot change column type: score: long -> int
```

エラーはコミット後にサーバ側から返るのではなく、PyIceberg の ValidationError としてクライアント側で発生しています。

列削除後は score がスキーマから消えました。

```
    [after delete_column]
      field-id=1   id           long       required
      field-id=2   full_name    string     optional
      field-id=3   email        string     optional
```

スキーマの履歴は次のとおりです。

```
--- スキーマの履歴
    テーブルが保持するスキーマ数: 5
      schema-id=0: ['id', 'name']
      schema-id=1: ['id', 'name', 'email']
      schema-id=2 (current): ['id', 'full_name', 'email']
      schema-id=3: ['id', 'full_name', 'email', 'score']
      schema-id=4: ['id', 'full_name', 'email', 'score']
```

5 回の変更に対して 5 個のスキーマが残っています。ここで 2 点、読み取れることがあります。

1 つは current が schema-id=2 になっている点です。`score` を削除した結果、列構成が rename 直後の schema-id=2 と一致したため、新しいスキーマを作らず既存のものが再利用されたとみられます（実装の詳細は未確認）。スキーマ ID は単調に進むとは限りません。

もう 1 つは schema-id=3 と 4 の列名が同一に見える点です。両者の違いは `score` の型（int と long）で、列名の一覧だけでは区別できません。型を含めて比較しないと同じスキーマに見えます。

6.5 ここから分かること

rename と列削除は、データファイルの書き直しを伴いません。TB 級のテーブルでも列名の変更はメタデータ操作で完結します。Hive テーブルで列名の変更が事実上できなかったのと対照的です。

一方、名前が単なるラベルであることには裏返しがあります。`a` を消して、あとから同じ名前 `a` の列を追加すると、新しい field ID が振られ、古いデータファイルの値は読めません。名前が同じでも別の列です。列の入れ替えを名前ベースで考えると認識を誤ります。

過去のスキーマがすべて保持されるのは、古いスナップショットを読むときに当時のスキーマが必要だからです。スキーマ変更を繰り返せばメタデータ JSON はその分大きくなります。頻繁なスキーマ変更は、スナップショットの蓄積と同じくメタデータの肥大要因になります。

型プロモーションはクライアント側で検証され、許可されない変更はコミットに至りません。ただし許可される範囲は format version によって拡張されており、どの変換が使えるかはテーブルの version に依存します。

前は 11. 基本的な CRUD と既定値、次は 13. パーティション進化と hidden partitioning です。

7 13. パーティション進化と hidden partitioning

samples/03_partition_evolution.py は、非パーティションのテーブルにパーティション仕様(spec)を 2 段階で追加し、その前後でデータを書きます。同一テーブル内に異なる spec で書かれたファイルが共存する様子と、クエリ側がパーティション列を意識しなくてよい仕組みを確認します。

Hive のパーティションはディレクトリ構造そのものだったため、後から変更するにはテーブルの作り直しが必要でした。Iceberg のパーティションはメタデータ上の定義で、変更は既存データに影響しません。

7.1 確かめたいこと

- パーティション仕様を変更したとき、既存データが書き換えられるか
- 古い spec と新しい spec が同一テーブル内に共存するか
- パーティション列に触れないクエリで枝刈りが効くか
- 古い spec で書かれたファイルのパーティション値がどう見えるか

7.2 背景

Iceberg のパーティションは、ソース列に変換(transform)を適用した結果として定義されます。`identity`、`day`、`bucket[N]`、`truncate[N]` などがあり、変換後の値は manifest ファイル内のパーティションタプルとして保持されます。

仕様は spec の進化について次のように述べています(Table Spec, Partition Evolution の項)。

> When evolving a spec, changes should not cause partition field IDs to change because the partition field IDs are used as the partition tuple field IDs in manifest files.

また、manifest の解釈については次のとおりです。

> Conversion uses the partition spec that was used to write the manifest file regardless of the current partition spec.

古い manifest は、それを書いた時点の spec で解釈されます。現在の spec で読み直すわけではないので、既存データを書き換える必要がありません。パーティションフィールドの ID が 1000 から始まる連番で、変更時にも既存分が維持されるのはこのためです。

hidden partitioning は、パーティション値を格納する物理列をテーブルスキーマに露出させない設計を指します。クエリはソース列(例では `ts`)に対する述語だけを書き、Iceberg が inclusive projection によってパーティション述語(`ts_day >= day(X)`)を導出し、枝刈りに使います。Hive では `WHERE ts >= X AND ts_day >= '2026-07-05'` のように論理述語とパーティション述語の両方を書く必要があり、片方を書き忘れるとフルスキャンになるか結果が誤りました。

7.3 処理の流れ

1. lab.events を event_id、region、ts の 3 列で、パーティションなしで作成します。
2. region の identity パーティションを追加します（spec 1）。

```
with tbl.update_spec() as us:
    us.add_field("region", IdentityTransform(), "region_part")
```

3. spec 1 の状態で 5 行（asia 3 行、europe 2 行）を append します。
4. ts の day transform を追加します（spec 2）。既存の region_part は残したまま、フィールドが 1 つ増えます。

```
with tbl.update_spec() as us:
    us.add_field("ts", DayTransform(), "ts_day")
```

5. spec 2 の状態で asia 3 行を append します。
6. tbl.inspect.files() でデータファイルごとの spec_id を確認します。

```
files = tbl.inspect.files()
df = files.select(["file_path", "spec_id", "record_count"]).to_pandas()
```

7. ts だけを条件にしたスキャンを実行し、パーティション列に触れずに絞り込めることを確認します。

```
result = tbl.scan(row_filter=f"ts >= '{day2.isoformat()}'").to_arrow()
```

8. tbl.inspect.partitions() でパーティションごとの状態を表示します。

7.4 実測結果

spec の変遷は次のとおりです。

```
--- 最初は非パーティションで作ります
    [initial] spec-id=0
      (パーティションなし)

--- region で identity パーティションを追加します
    [spec 1] spec-id=1
      field-id=1000 source-id=2 name=region_part transform=identity

--- パーティションを進化させます: ts の day transform を追加
    [spec 2] spec-id=2
      field-id=1000 source-id=2 name=region_part transform=identity
      field-id=1001 source-id=3 name=ts_day transform=day
```

region_part のパーティションフィールド ID は spec 1 と spec 2 の両方で 1000 のままです。新しく追加された ts_day に 1001 が振られています。source-id はテーブルスキーマ側の field ID で、region が 2、ts が 3 に対応します。

テーブルは過去の spec もすべて保持しています。

```
    テーブルが保持する spec 一覧:
      spec-id=0: []
      spec-id=1: ['region_part=identity']
      spec-id=2 (default): ['region_part=identity', 'ts_day=day']
```

データファイルごとの spec_id です。

```
                                       file_path  spec_id  record_count
00000-0-0ad8c5a8-5aeb-4e43-bf6e-bcdab0350d9b.parquet        2             3
00000-0-f031aacc-a68a-4e5a-9553-43785f7dd2d7.parquet        1             2
00000-0-cc7e947e-8c17-47b4-89cf-c341ea95a580.parquet        1             3
```

spec 1 で書いた 2 ファイル(2 行と 3 行)はそのまま spec_id=1 として残り、spec 2 で書いた 1 ファイル(3 行)だけが spec_id=2 です。spec の変更時に既存ファイルの書き直しは起きていません。

ts のみを条件にしたスキャンの結果です。

```
--- hidden partitioning: クエリにパーティション列を書かない
    ts >= 2026-07-05 で 3 行
```

クエリで指定したのは ts に対する述語だけで、ts_day には触れていません。全 8 行のうち 2026-07-05 以降の 3 行が返りました。

パーティションごとの状態は次のとおりです。

```
                                 partition  spec_id  record_count  file_count
{'region_part': 'asia', 'ts_day': 2026-07-05}        2             3           1
    {'region_part': 'europe', 'ts_day': None}        1             2           1
      {'region_part': 'asia', 'ts_day': None}        1             3           1
```

spec 1 で書かれたファイルの ts_day は None です。書かれた時点の spec に ts_day が存在しなかったので、パーティションタプルにも値がありません。同じ region_part='asia' でも、spec_id が違えば別のパーティションエントリとして数えられています。

7.5 ここから分かること

パーティション設計は後から変えられます。日次パーティションが粗すぎる、あるいは細かすぎると分かった時点で spec を足せば、以降の書き込みから新しい粒度が適用されます。既存データを作り直す必要はありません。ただし、既存データの物理配置が新しい spec に従うわけでもないので、過去分にも新しい粒度を効かせたいなら再書き込み（rewrite）が別途必要です。

古い spec のファイルでパーティション値が None になる点は、枝刈りの効き方に影響します。`ts_day` が未定義のファイルは `ts_day` を使った絞り込みでは除外できず、ファイル単位の統計（列の min/max）に頼ることになります。spec を変えた直後は、変更前のデータに対する枝刈りが期待どおりに効かない可能性があります。

hidden partitioning によって、クエリ側はパーティション設計を知らなくて済みます。裏を返せば、パーティション設計を変えても既存のクエリを書き換える必要がありません。Hive でパーティション列がクエリ本文に埋め込まれていたのと比べると、設計変更のコストがクエリ側に波及しなくなっています。

前は 12. スキーマ進化と field ID、次は 14. タイムトラベルと branch / tag です。関連する調査内容は姉妹リポジトリの調査報告書にまとめられています。

8 14. タイムトラベルと branch / tag

Iceberg のテーブルは、コミットのたびにスナップショットを積み上げます。古いスナップショットはコミット後も残り続けるため、過去の任意の時点の状態をそのまま読めます。この仕組みの上に、名前付き参照である branch と tag が乗っています。

ここでは `samples/04_time_travel.py` を使い、3 世代のスナップショットを作ってから、ID 指定の読み取り、tag と branch の作成、ロールバックまでを一通り実行します。あわせて、tag を打つとスナップショットの寿命が延びるという運用上の副作用も確認します。

8.1 確かめたいこと

- スナップショット ID を指定すると過去の状態が読めること
- tag と branch がテーブルの refs にどう現れるか
- tag / branch から参照されたスナップショットは expire の対象外になること
- ロールバック後、捨てられたコミットが履歴上どう見えるか

8.2 背景

Iceberg のテーブルメタデータは `refs` というマップを持ち、名前を snapshot ID に対応づけます。branch は移動する参照、tag は固定された参照です。仕様には次の記述があります。

> There is always a `main` branch reference pointing to the current-snapshot-id even if the refs map is null.
>
> The `main` branch never expires.

つまり `main` は特別扱いで、明示的に refs へ書かれていなくても存在し、期限切れにもなりません。

もうひとつ、branch はデータの分岐であってスキーマの分岐ではありません。仕様は次のように述べています。

> the schema tracked for a table is valid across all branches

スキーマはテーブル全体で 1 つです。ブランチごとに異なるスキーマを持たせることはできません。

保持の観点では、branch や tag から参照されているスナップショットは expire されません。tag は「消えてほしくないスナップショットに錨を打つ」操作であり、同時に「そのスナップショットが参照するデータファイルを解放できなくする」操作でもあります。

8.3 処理の流れ

1. lab.versioned テーブルを作り直し、3 回に分けて append します。各 append の間に time.sleep(1) を挟み、コミットのタイムスタンプを分離します。
2. 各世代の snapshot ID を控えたうえで、inspect.snapshots() で履歴を表示します。
3. スナップショット ID を指定してスキャンします。

```
for label, sid in (("世代1", snap1), ("世代2", snap2), ("世代3", snap3)):
    n = tbl.scan(snapshot_id=sid).to_arrow().num_rows
```

4. 世代 2 に tag を打ち、inspect.refs() で refs の中身を確認します。

```
with tbl.manage_snapshots() as ms:
    ms.create_tag(snapshot_id=snap2, tag_name="release-2026-07")
```

5. 同じく世代 2 を指す audit ブランチを作ります。Write-Audit-Publish は、この監査用ブランチに書いて検証してから main へ fast-forward する運用です。ただし fast-forward は Spark の procedure であり、PyIceberg には該当機能がありません。
6. 世代 1 へロールバックし、inspect.history() で is_current_ancestor を確認します。

```
tbl.manage_snapshots().rollback_to_snapshot(snap1).commit()
```

8.4 実測結果

3 世代のスナップショットは次のようになりました。

```
--- 3 回に分けてデータを書き、3 つのスナップショットを作ります
    世代 1: snapshot_id=7742509384658575965, 行数=1
    世代 2: snapshot_id=3695037040879739491, 行数=3
    世代 3: snapshot_id=1545959734487094455, 行数=4
```

履歴では parent_id が 1 つ前の snapshot ID を指しており、コミットが線形に連なっていることが読み取れます。

```
--- スナップショット履歴
           committed_at         snapshot_id    parent_id operation
2026-07-18 13:49:32.679 7742509384658575965          NaN    append
2026-07-18 13:49:33.789 3695037040879739491 7.742509e+18    append
2026-07-18 13:49:34.894 1545959734487094455 3.695037e+18    append
```

スナップショット ID を指定した読み取りは、それぞれの時点の行数を返します。

```
--- スナップショット ID でタイムトラベル
    世代 1 (id=7742509384658575965): 1 行
    世代 2 (id=3695037040879739491): 3 行
    世代 3 (id=1545959734487094455): 4 行
```

tag と branch を作ったあとの refs は次のとおりです。main は最新の世代 3 を、audit ブランチと release-2026-07 タグは世代 2 を指しています。

```
           name   type         snapshot_id  max_reference_age_in_ms  min_snapshots_to_keep  max_snapshot_ag
           main BRANCH 1545959734487094455                      NaN                    NaN
          audit BRANCH 3695037040879739491                      NaN                    NaN
release-2026-07    TAG 3695037040879739491                      NaN                    NaN
```

保持設定の列がすべて NaN であることに注意してください。参照ごとの保持ポリシー（max-reference-age-ms など）は指定していないため、これらの参照は既定で無期限に残ります。

ロールバックは行数を 4 行から 1 行に戻しました。

```
--- 世代 1 にロールバックします
    ロールバック前: 4 行
    ロールバック後: 1 行
```

ロールバック後の履歴には、捨てられたコミットが is_current_ancestor=False として残っています。

```
        made_current_at         snapshot_id    parent_id  is_current_ancestor
2026-07-18 13:49:32.679 7742509384658575965          NaN                 True
2026-07-18 13:49:33.789 3695037040879739491 7.742509e+18                False
2026-07-18 13:49:34.894 1545959734487094455 3.695037e+18                False
2026-07-18 13:49:35.140 7742509384658575965          NaN                 True
```

最終行は「世代 1 を再び current にした」というエントリです。ロールバックはスナップショットの削除ではなく、参照の付け替えとして記録されます。世代 2・世代 3 のスナップショットは消えていません。しかもこの実行では世代 2 に tag と branch が付いているため、仮に expire を回しても世代 2 は残ります。

8.5 ここから分かること

タイムトラベルが成立するのは、古いデータファイルがスナップショットから参照され続けているからです。裏返すと、expire_snapshots を実行しない限りストレージは解放されません。ロールバックしても容量は減りません。

tag の運用コストは見落とされやすい点です。tag を 1 つ打つと、そのスナップショットと、それが参照するデータファイル一式が保持対象になります。参照ごとの保持ポリシーを設定しなければ、これは無期限です。

マネージドサービスではさらに影響が広がる場合があります。Amazon S3 Tables の公式ドキュメントは、テーブルにユーザー定義の tag や branch が存在すると、そのテーブル全体で snapshot

management が失敗すると述べています。tag や branch 側の retention 期間が短くても同様とされています。この場合、自動メンテナンスの停止は例外としては現れず、課金の増加としてしか観測できません。

WAP を PyIceberg だけで完結させることはできません。branch の作成と書き込みまではできますが、main への fast-forward は Spark procedure です。ブランチ運用を前提にするなら、Spark を含む構成を最初から想定しておく必要があります。PyIceberg 側の機能範囲については 17. PyIceberg の制約を実証する を参照してください。メタデータ上で snapshot と refs がどう格納されているかは 15. メタデータ三層構造を覗く で扱います。

9 15. メタデータ三層構造を覗く

Iceberg のテーブルは、カタログが持つ 1 本のポインタから始まり、metadata.json、manifest list、manifest、data file という順に辿れる構造になっています。この階層があるおかげで、スキャン計画はデータファイルを開かずに大半を切り落とせます。

ここでは samples/05_metadata.py を使い、実際のテーブルからこの各層を順に取り出して表示します。あわせて、プランニングが使う統計情報が実際にどう格納されているかを確認します。

9.1 確かめたいこと

- カタログが保持しているのが metadata.json への 1 本のポインタだけであること
- スナップショットごとに manifest list が 1 つ対応すること
- manifest が data file の一覧と要約統計を持つこと
- 列統計（lower_bound / upper_bound）が実際にどんな値で入っているか
- manifest entry の sequence_number が何を表すか

9.2 背景

構造は次のように連なります。

```
catalog → metadata.json → manifest list → manifest → data file
```

カタログが管理する状態は、テーブルごとの metadata.json の位置だけです。コミットとは、このポインタを新しい metadata.json へアトミックに差し替える操作にほかなりません。ポインタ差し替えの実際のプロトコルは 16. REST API を直接叩く で扱います。

スキャン計画は段階的に絞り込みます。第 1 段は manifest list に記録された各 manifest のパーティション要約統計（field_summary）だけを見て、manifest ファイル自体を開かずにスキップ判定します。第 2 段で残った manifest を開き、第 3 段で data file ごとの列統計を使ってファイル単位のスキップ（file skipping）を行います。

9.3 処理の流れ

1. `lab.metadata_demo` を作り直し、5 行ずつ 3 回 append します。append を分けることで manifest が複数できます。
2. 第 1 層として `tbl.metadata` の主要フィールドと `tbl.metadata_location` を表示します。
3. `inspect.metadata_log_entries()` で、過去の metadata.json の履歴を表示します。

4. 第 2 層として、各スナップショットの manifest_list のパスを表示します。

```
for snap in md.snapshots:
    print(f"    sequence-number = {snap.sequence_number}")
    print(f"    manifest-list   = .../{snap.manifest_list.rsplit('/', 1)[-1]}")
```

5. 第 3 層として inspect.manifests() で manifest 一覧を、続いて inspect.files() で data file 一覧を表示します。
6. readable_metrics を展開し、列ごとの統計を人が読める形で確認します。

```
rm = files.select(["file_path", "readable_metrics"]).to_pylist()
for col, stats in rm[0]["readable_metrics"].items():
    ...
```

7. 最後に inspect.entries() で manifest entry の status と sequence_number を表示します。

9.4 実測結果

第 1 層の metadata.json は次の状態でした。書き込み 3 回に対してスナップショットが 3 個、refs は main のみです。

```
    location            = s3://warehouse/lab_catalog/lab/metadata_demo
    table-uuid          = 4bf21d74-231a-4c1a-acaa-70912f1b4d25
    format-version      = 2
    last-sequence-number= 3
    current-snapshot-id = 9056089237084054718
    schemas             = 1 個
    partition-specs     = 1 個
    snapshots           = 3 個
    refs                = ['main']
```

カタログが指しているファイルは 1 つだけです。

```
      s3://warehouse/lab_catalog/lab/metadata_demo/metadata/00003-b5aa2cfc-6d34-40ba-ad39-
2218c096c7fb.metadata.json
```

metadata の履歴は 4 エントリありました。1 行目はテーブル作成時点で、まだスナップショットがないため latest_snapshot_id が NaN です。

```
              timestamp  latest_snapshot_id  latest_sequence_number
2026-07-18 13:49:36.920                 NaN                     NaN
2026-07-18 13:49:37.221        6.202441e+18                     1.0
2026-07-18 13:49:37.320        7.230180e+18                     2.0
2026-07-18 13:49:37.406        9.056089e+18                     3.0
```

第 2 層では、3 つのスナップショットにそれぞれ別の manifest list が対応していました。sequence-number は 1, 2, 3 と増加し、parent が連鎖しています。

```
snapshot-id     = 6202440951381350097
parent          = None
sequence-number = 1
operation       = Operation.APPEND
manifest-list   = .../snap-6202440951381350097-0-c0f71b99-dfa5-4c08-b891-60ef86c219dc.avro
```

第 3 層の manifest は 3 つ、いずれも `added_data_files_count` が 1 です。append ごとに manifest が 1 つ、data file が 1 つ作られたことが分かります。

```
                                      path  length  partition_spec_id  added_data_files_count  existing_d
ec39da35-6253-4d8f-8629-b738f2fbd8ca-m0.avro    4436                  0                       1
e8c8624d-fad3-40c7-9e0f-d4e51e89ade5-m0.avro    4438                  0                       1
c0f71b99-dfa5-4c08-b891-60ef86c219dc-m0.avro    4434                  0                       1
```

data file は 5 行で 1746〜1753 バイトでした。manifest（4434〜4438 バイト）のほうが、それが記述する data file より大きいという逆転が起きています。小さい書き込みを繰り返すとメタデータの比率が跳ね上がることが、この規模でも見て取れます。

```
                                         file_path file_format  record_count  file_size_in_bytes
00000-0-ec39da35-6253-4d8f-8629-b738f2fbd8ca.parquet     PARQUET             5                1747
00000-0-e8c8624d-fad3-40c7-9e0f-d4e51e89ade5.parquet     PARQUET             5                1753
00000-0-c0f71b99-dfa5-4c08-b891-60ef86c219dc.parquet     PARQUET             5                1746
```

列統計は次のとおりです。

```
  id:
    column_size          = 123
    value_count          = 5
    null_value_count     = 0
    nan_value_count      = None
    lower_bound          = 200
    upper_bound          = 204
  category:
    lower_bound          = A
    upper_bound          = C
  amount:
    lower_bound          = 2.0
    upper_bound          = 42.0
  ts:
    lower_bound          = 2026-07-03 00:00:00
    upper_bound          = 2026-07-03 04:00:00
```

id が 200〜204、ts が 2026-07-03 の 4 時間分に収まっています。これは 3 回目の append（batch=2）で書かれたファイルです。述語 id < 100 が来れば、このファイルは lower_bound を見た時点で開かずに除外できます。

manifest entry は 3 件、いずれも status=1（ADDED）でした。

```
 status         snapshot_id  sequence_number  file_sequence_number
      1 9056089237084054718                3                     3
      1 7230179648642667438                2                     2
      1 6202440951381350097                1                     1
```

9.5 ここから分かること

コミットの実体は metadata.json のポインタ差し替えです。カタログ側が保持する状態はこの 1 つだけで、それ以外の情報はすべてオブジェクトストレージ上のファイルに載っています。カタログを差し替えても、データとメタデータの構造は変わりません。

file skipping は列統計に依存します。ここで効いてくる既定値が 2 つあります。

- write.metadata.metrics.default = truncate(16) --- 文字列やバイナリの境界値は 16 文字で切り詰められます。URL や UUID 文字列のように共通プレフィックスが長い列では、境界値が実質同一になりプルーニングが効きません。たとえば https://example.com/a/... と https://example.com/z/... は、先頭 16 文字がどちらも https://example. なので、統計上は同じ範囲に見えます。ファイルごとの lower/upper がすべて一致するため、どの述語を投げても 1 ファイルも枝刈りできません。
- write.metadata.metrics.max-inferred-column-defaults = 100 --- 101 列目以降には既定でメトリクスが収集されません。ワイドテーブルで後方の列に述語をかけると、全ファイル読みになります。

metadata.json の蓄積も運用上の論点です。write.metadata.previous-versions-max の既定は 100 で、write.metadata.delete-after-commit.enabled の既定は false です。何もしなければ metadata JSON は溜まり続けます。さらに、追跡上限の 100 を溢れた分は untracked になるため、後から削除を有効化しても対象になりません。

sequence_number はコミット順を表し、delete ファイルの適用判定に使われます。position delete は「データ seq <= delete seq」、equality delete は「データ seq < delete seq」（厳密に小さい）で適用されます。この非対称性が、同一コミット内で追加した行を position delete で消せる理由です。

小ファイルとメタデータの膨張は compaction で解消しますが、PyIceberg には compaction がありません。詳細は 17. PyIceberg の制約を実証するを参照してください。運用面の整理は姉妹リポジトリの調査報告書（https://dobachi.github.io/iceberg-research/ ）にまとめてあります。

10 16. REST API を直接叩く

Iceberg REST Catalog は HTTP の API で、その仕様は OpenAPI（API の形を機械可読な定義ファイルで記述する書式）で公開されています。PyIceberg は内部でこの API を呼んでいますが、クライアントライブラリを通すと、コミットが何を送って何が返るのかが見えなくなります。

ここでは `samples/06_rest_api_raw.py` を使い、httpx で直接エンドポイントを叩きます。トークン取得、`GET /v1/config`、loadTable、updateTable のコミットプロトコル、エラー形式の順に確認します。特に updateTable の requirements / updates の構造と、409 が返る条件はこのラボの中核です。

10.1 確かめたいこと

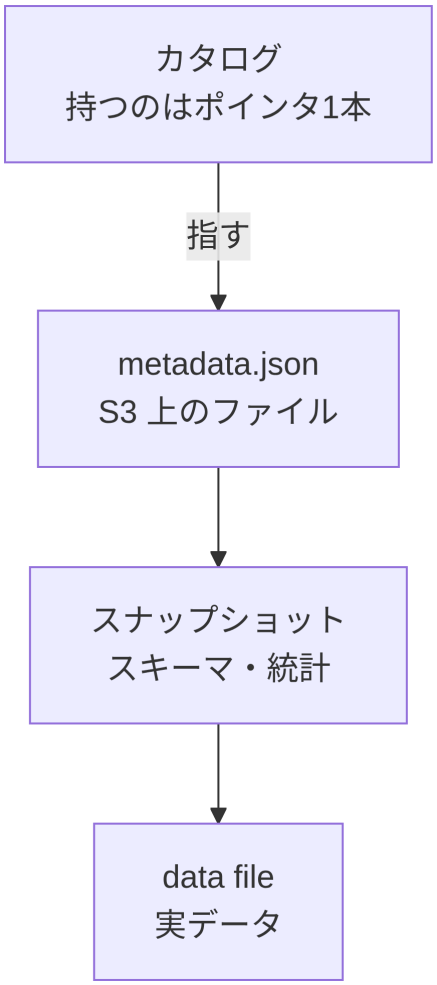
- `GET /v1/config` の defaults / overrides / endpoints をどう解釈するか
- loadTable のレスポンスに何が含まれるか（credential vending を含む）
- updateTable の requirements（表明）と updates（変更）の構造
- 意図的に古い snapshot ID を表明したとき 409 が返ること
- 409 と 500 の意味の違い
- エラーレスポンスの形式

10.2 そもそもカタログは何をしているのか

先に全体像を整理します。

Iceberg のテーブルの実体は、オブジェクトストレージ上のファイル群です（15. メタデータ三層構造を覗くで見たとおり）。データも、スキーマも、スナップショットの履歴も、すべて S3 上のファイルに書かれています。

ではカタログは何を持っているのかというと、**「このテーブルの最新の metadata.json はどれか」という 1 本のポインタだけ**です。



この構造から、カタログに必要な機能が決まります。

- **ポインタを読む** --- テーブルを開く（loadTable）
- **ポインタを差し替える** --- コミットする（updateTable）
- **ポインタの一覧を管理する** --- namespace やテーブルの作成・一覧・削除

つまり **Iceberg のコミットとは、カタログが持つポインタを新しい metadata.json に差し替えること**です。REST Catalog は、この操作を HTTP で行うための取り決めにすぎません。

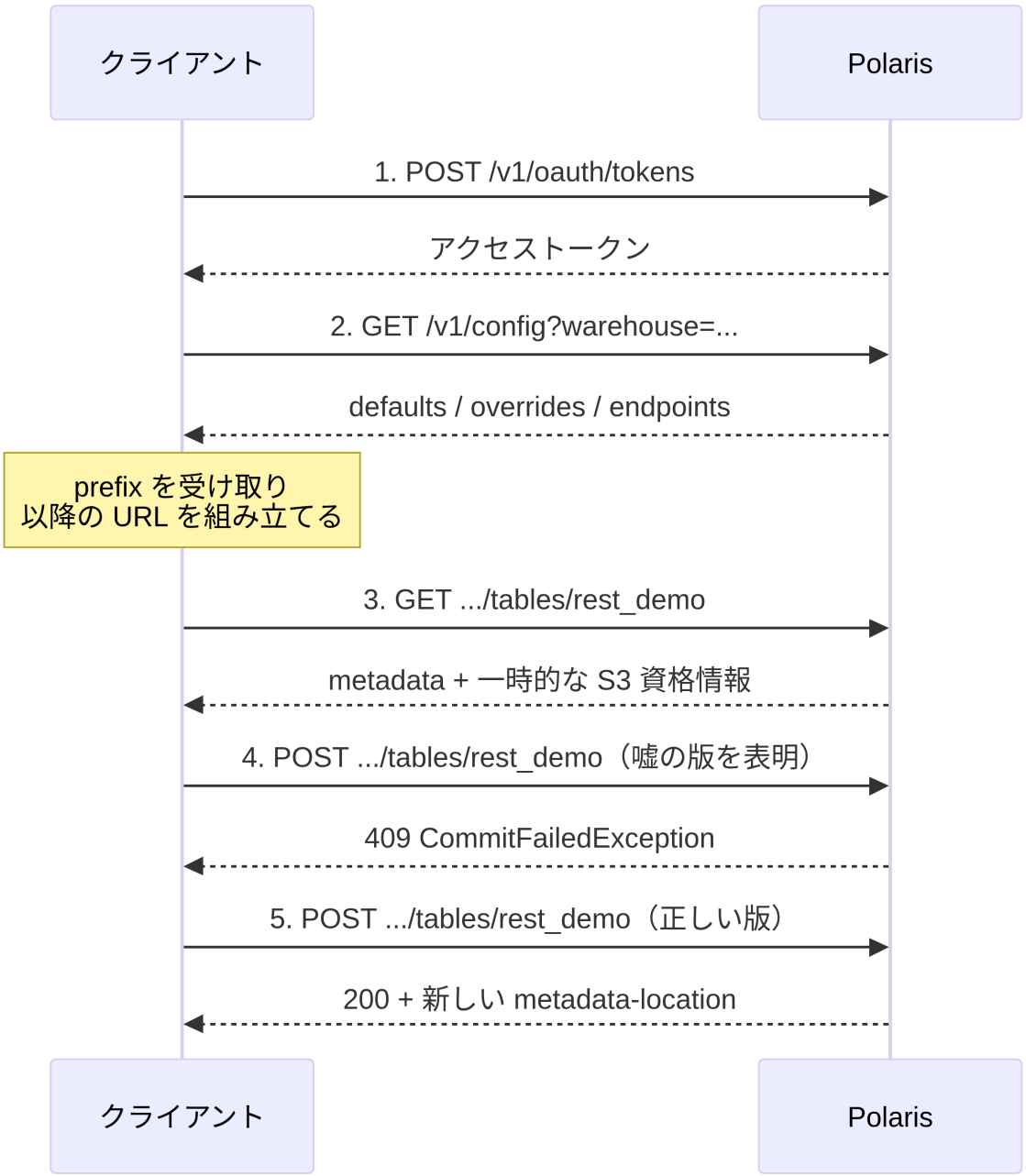
10.2.1 なぜ HTTP なのか

以前は、カタログを Hive Metastore や JDBC で直接触るのが一般的でした。この方式では、各エンジン（Spark、Trino、Flink…）がカタログの実装ライブラリを抱え込むことになります。

REST にすると、サーバ側が仕様どおりの HTTP を話しさえすれば、クライアントは実装を知らずに済みます。カタログを差し替えても、クライアント側のコードは変わりません。この「差し替え可能性」が REST カタログの主眼です。

10.3 全体の流れ

このサンプルが叩く順序です。



1 から 3 までは準備で、**4 と 5 がこの実験の核心**です。同じリクエストで、表明する「版」だけを変えて成否が分かれます。

10.4 背景

10.4.1 認証: なぜ最初にトークンを取るのか

HTTP には状態がないため、リクエストごとに「自分が誰か」を示す必要があります。毎回パスワードを送るのは危険なので、OAuth2 では次の 2 段階を踏みます。

1. **一度だけ**、client ID と client secret を送ってトークンを受け取る
2. 以降は `Authorization: Bearer <トークン>` ヘッダで、そのトークンを提示する

トークンには有効期限があり、漏れても影響が期限内に限られます。ここで使っている `client_credentials` は「人間ではなくプログラムが自分自身として認証する」ための方式です。

10.4.2 設定のネゴシエーション

仕様は `GET /v1/config` について次のように述べています。

> All REST clients should first call this route

なぜ最初に呼ぶ必要があるのかというと、**接続先ごとに設定が違い、その一部はサーバが決めるから**です。クライアントは自分の設定とサーバの設定を合成してから動き出します。

レスポンスには `defaults` と `overrides` が含まれます。名前のとおりの役割です。

フィールド	意味	クライアントは上書きできるか
`defaults`	サーバからの**提案**。クライアントが何も指定しなければこれを使う	できる
`overrides`	サーバの**指定**。サーバ側の都合で強制したい値	できない

適用順は defaults → クライアント設定 → overrides で、後に来たものが勝ちます。つまりクライアントは defaults を上書きできますが、overrides には逆らえません。

10.4.2.1 prefix とは何か

`overrides` でよく配られるのが `prefix` です。これは**以降のリクエストのパスに差し込む文字列**です。

```
prefix なし: /v1/namespaces/lab/tables/trips
prefix あり: /v1/lab_catalog/namespaces/lab/tables/trips
                  ~~~~~~~~~~~
```

1 台のサーバが複数のカタログを提供するとき、どのカタログ宛てかをパスで区別するために使われます。**クライアントが勝手に決める値ではなく、サーバから配られる値**である点が重要です。`GET /v1/config` を呼ばずに URL を組み立てると、以降のリクエストが全部パスを外します。

10.4.2.2 「フィールドがあること」自体が意味を持つ

`endpoints` と `idempotency-key-lifetime` は、少し変わった使われ方をします。**値ではなく、フィールドが存在するかどうかがサーバの対応可否を表します。**

`endpoints` はサーバが対応しているエンドポイントの一覧です。このフィールドが無い場合、クライアントは仕様が定めるデフォルトセット（13 個）だけを仮定します。view やスキャンプランニング、credential vending はそこに含まれません。つまり「宣言が無い＝古い最小構成とみなす」という約束です。

`idempotency-key-lifetime` も同じ形です。仕様はこう述べています。

> If absent, clients MUST assume idempotency is not supported.

10.4.2.2.1 冪等（べきとう）とは

同じ操作を何回繰り返しても、1 回だけ実行したのと同じ結果になる性質です。

これがコミットで問題になるのは、**リクエストを送ったあと応答が返ってこないとき**です。ネットワークが切れた場合、クライアントには次の区別が付きません。

- サーバに届かなかった（コミットされていない）
- サーバは処理したが、応答が返る途中で切れた（コミット済み）

前者ならリトライすべきで、後者でリトライすると**同じデータをもう一度追加**してしまいます。

`Idempotency-Key` ヘッダは、この曖昧さを解消するための仕組みです。クライアントがリクエストに一意な鍵を付けておくと、サーバは同じ鍵のリクエストを 2 回目以降は処理せず、1 回目の結果を返します。これでリトライが安全になります。

`idempotency-key-lifetime` は、サーバがその鍵を覚えておく期間を表します。**このフィールドが無いサーバは鍵を覚えてくれない**ので、クライアントはリトライの安全性を自前で確保しなければなりません。

10.4.3 コミットプロトコル

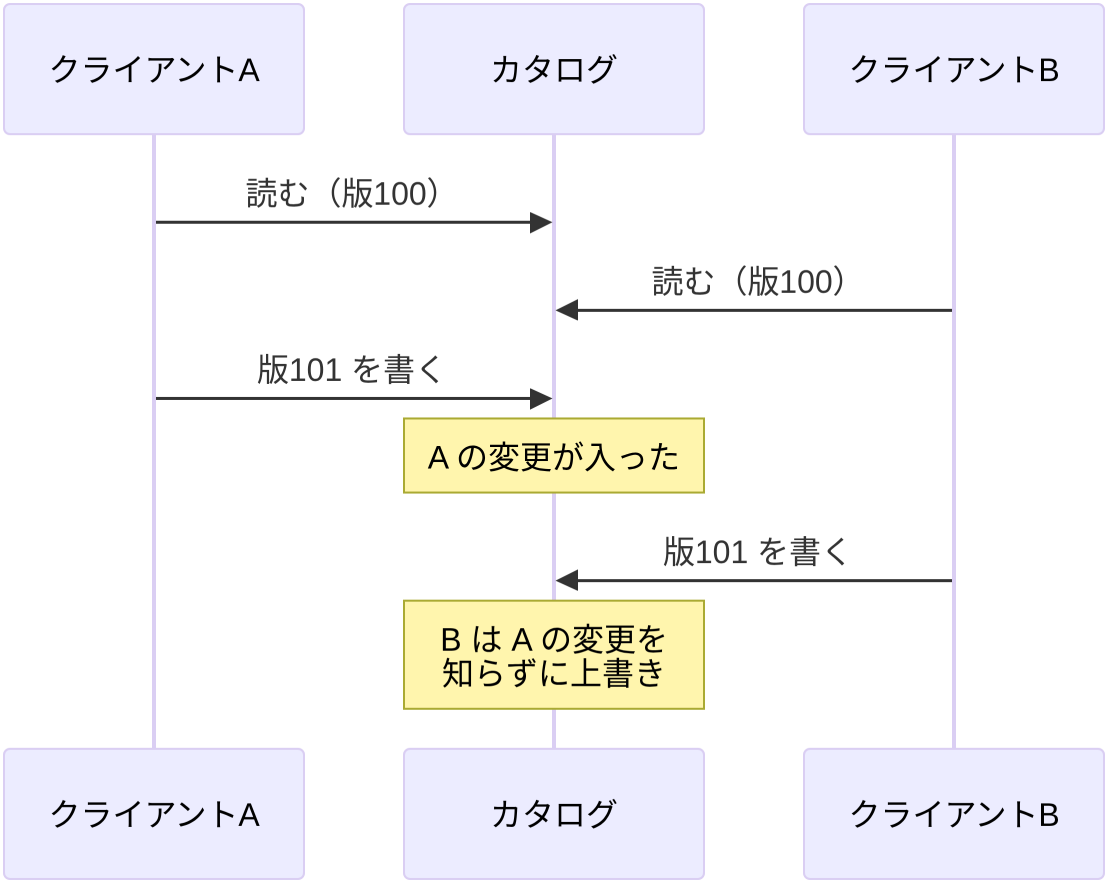
`POST /v1/{prefix}/namespaces/{namespace}/tables/{table}`（updateTable）のボディは、大きく 2 つの配列でできています。

- `requirements` --- コミット前に成り立っていなければならない条件の表明。例: `assert-table-uuid`、`assert-ref-snapshot-id`
- `updates` --- 適用したい変更。例: `set-properties`、`add-snapshot`、`set-snapshot-ref`

サーバは requirements を現在のテーブル状態と突き合わせ、すべて満たされる場合にだけ updates を適用します。requirements は「自分が読んだ版はこれだ」という宣言であり、その版が変わっていればコミットは拒否されます。

10.4.3.1 なぜこの形なのか

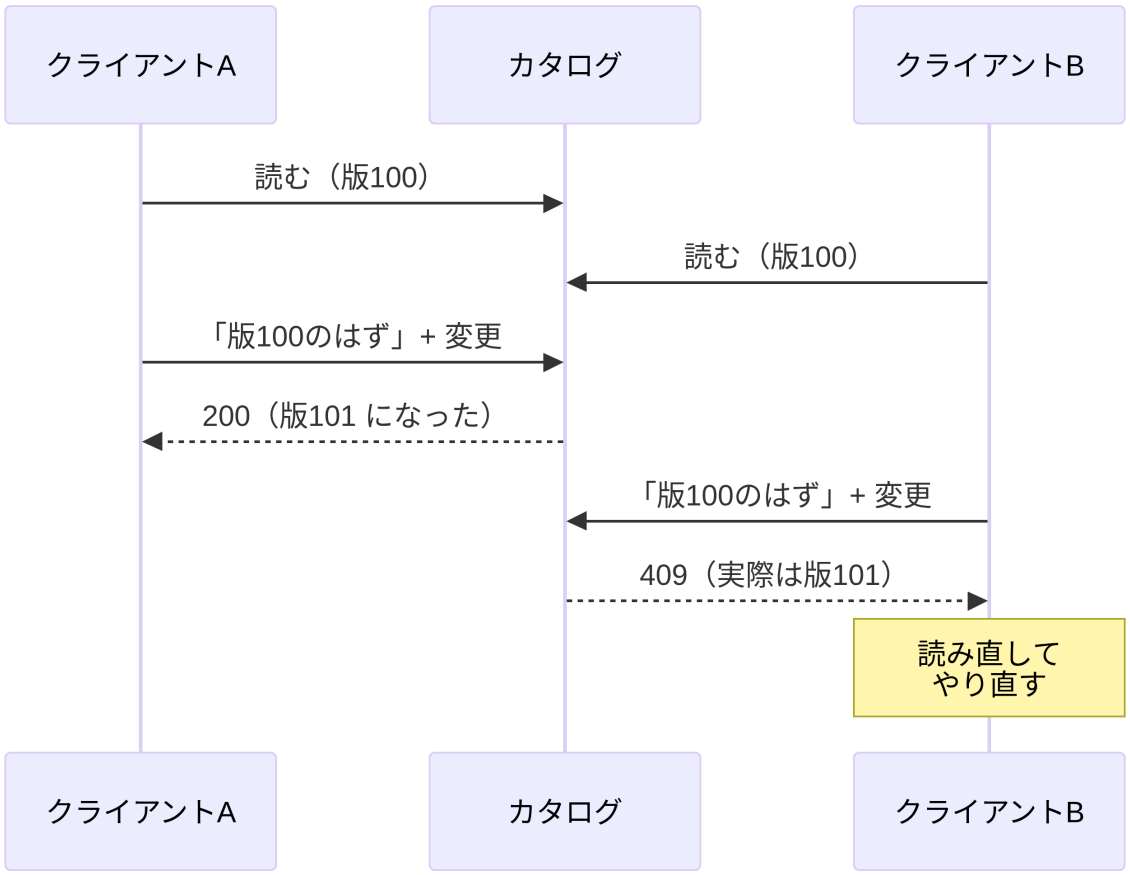
複数のクライアントが同じテーブルに同時に書くと、素朴な実装では**あとから書いたほうが前の変更を消してしまいます**。



これを防ぐ方法は 2 つあります。ひとつは**ロック**で、書く前にテーブルを占有します。確実ですが、ロックの管理が必要になり、持ったまま落ちたクライアントの後始末も要ります。

Iceberg が採るのはもうひとつの方法、**楽観的並行制御**です。ロックを取らずに書きに行き、書く瞬間に「自分が読んだときから変わっていないこと」を条件として付けます。変わっていれば失敗するので、読み直してやり直します。

これが compare-and-swap（CAS）です。requirements が「比較」、updates が「交換」にあたります。



衝突がまれであれば、ロックの管理コストを払わずに済みます。「楽観的」と呼ばれるのは、**衝突しない前提で進み、衝突したときだけ対処する**からです。

10.4.3.2 requirements と updates の例

主なものを挙げます。

requirements（表明）	意味
`assert-table-uuid`	相手が同じテーブルであること。削除して同名で作り直された場合に検出できる
`assert-ref-snapshot-id`	指定したブランチが、自分の見た版を指していること
`assert-current-schema-id`	スキーマが変わっていないこと

updates（変更）	意味
`set-properties`	テーブルプロパティを設定する
`add-snapshot`	新しいスナップショットを追加する
`set-snapshot-ref`	ブランチやタグの指す先を変える

データを追加するコミットは、実際には `add-snapshot`（スナップショットを足す）と `set-snapshot-ref`（main をそこに向ける）の組み合わせになります。

安全性の要は、未知の要素を無視しないことです。仕様は次のように述べています。

> Server implementations are required to fail with a 400 status code if any unknown updates or requirements are received.

未知の requirement を黙って読み飛ばす実装があると、クライアントは検証されたつもりでコミットが通ってしまいます。400 での失敗が必須とされているのはこのためです。

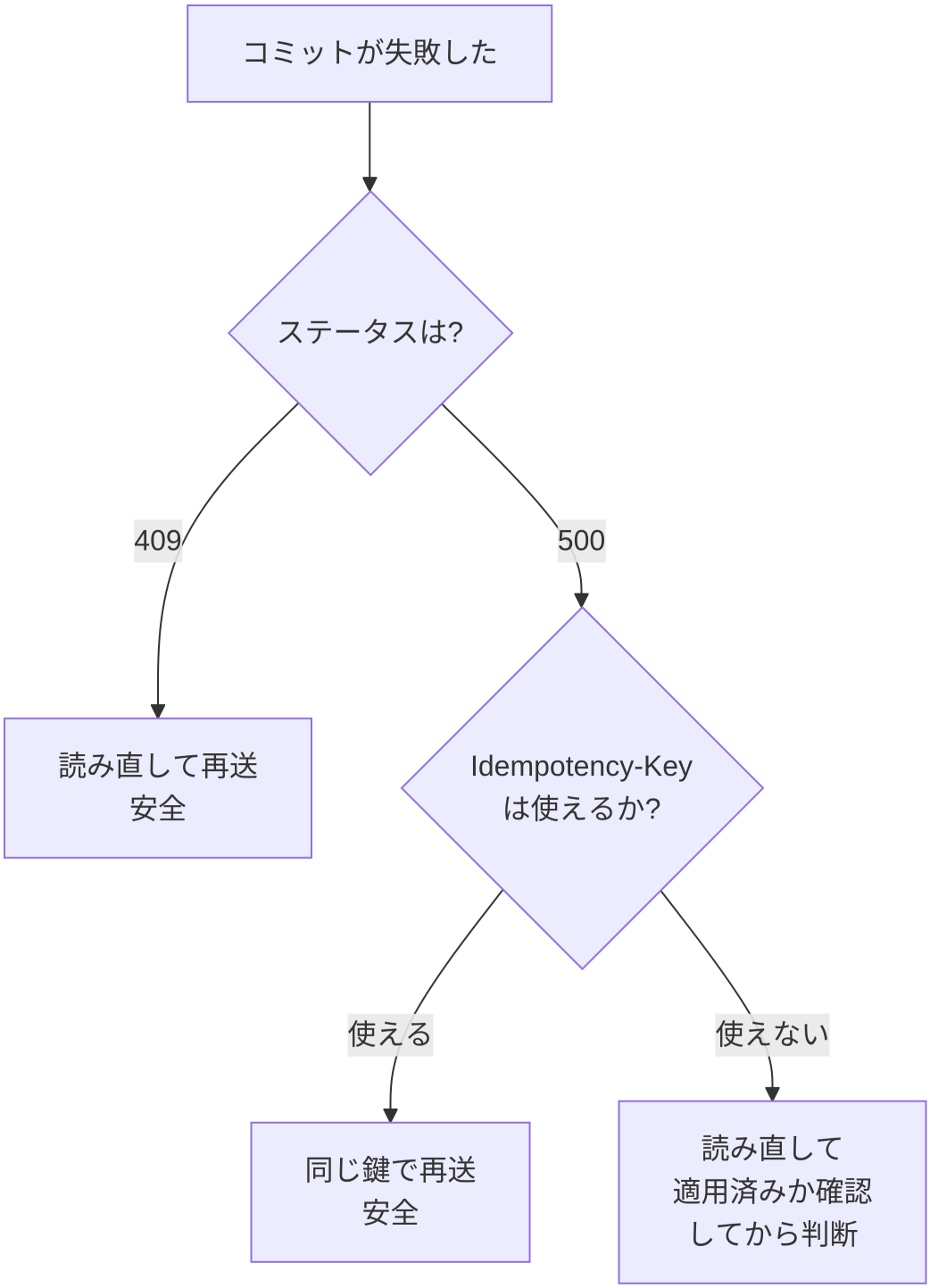
10.4.4 コミットが失敗したときの 2 種類

コミットの失敗には、意味がまったく違う 2 種類があります。**この区別を誤ると、データを壊します。**

ステータス	型	意味	リトライ
409	`CommitFailedException`	requirements が満たされなかった。**コミットは確実に適用されていない**	安全。読み直して再送する
500	`CommitStateUnknownException`	サーバが結果を確定できなかった。**適用されたか不明**	危険。そのまま再送してはいけない

409 は「あなたの見ていた版は古い」という明確な拒否なので、状態は変わっていません。読み直して requirements を作り直し、送り直せば済みます。

500 が厄介なのは、**成功したかもしれない**ことです。たとえばカタログが metadata.json のポインタを書き換えたあと、応答を返す前に落ちた場合がこれにあたります。ここで単純にリトライすると、同じスナップショットをもう一度追加してしまいます。データが二重に入るということです。



先ほどの `idempotency-key-lifetime` が効いてくるのがここです。サーバが対応していれば同じ鍵で送り直せば済みますが、対応していなければ、クライアントが自分でテーブルを読み直し、自分のコミットが入っているかどうかを確かめる必要があります。

10.4.4.1 409 には 2 つの意味がある

紛らわしいのですが、**同じ 409 が別の場面でも使われます**。

場面	型	意味	リトライ
namespace / テーブルの作成	`AlreadyExistsException`	すでに存在する	無意味（何度送っても同じ）
updateTable	`CommitFailedException`	コミット競合	有効（読み直せば通る）

ステータスコードだけで分岐すると、「すでに存在する」に対して延々とリトライする実装になります。**エラーボディの `type` を見て判断する**必要があります。

10.5 処理の流れ

1. `POST /v1/oauth/tokens` に client_credentials でトークンを要求します。このエンドポイントは仕様上 DEPRECATED for REMOVAL です（Iceberg Java 1.6.0 以降で非推奨、2.0 で仕様から削除予定）。本番では外部 IdP を使い `oauth2-server-uri` を明示的に設定します。
2. `GET /v1/config` に `warehouse` を付けて呼び、`overrides.prefix` を取り出して以降のベース URL を組み立てます。
3. namespace を作成し、既存なら 409 を確認します。
4. PyIceberg でテーブル `lab.rest_demo` を用意したうえで、loadTable を叩きます。credential vending を要求するヘッダを付けます。

```
r = client.get(
    f"{api}/namespaces/{c.NAMESPACE}/tables/{table_name}",
    headers={**auth, "X-Iceberg-Access-Delegation": "vended-credentials,remote-signing"},
)
```

仕様は次のように述べており、このヘッダは要求であって保証ではありません。

> The server may choose to supply access via any or none of the requested mechanisms

5. わざと存在しない snapshot ID を表明して updateTable を投げ、409 を起こします。

```
req = {
    "identifier": {"namespace": [c.NAMESPACE], "name": table_name},
    "requirements": [
        {"type": "assert-table-uuid", "uuid": table_uuid},
        {"type": "assert-ref-snapshot-id", "ref": "main", "snapshot-id": bogus_snapshot},
    ],
    "updates": [
        {"action": "set-properties", "updates": {"lab.marker": "should-not-apply"}}
    ],
}
```

6. requirements の snapshot-id を正しい値に直して再送し、成功を確認します。
7. 存在しないテーブルを loadTable してエラー形式を確認します。

10.6 実測結果

`GET /v1/config` のレスポンスは次のとおりでした（endpoints は抜粋）。

```
{
  "defaults": {
    "default-base-location": "s3://warehouse/lab_catalog"
  },
  "overrides": {
    "namespace-separator": "%1F",
    "prefix": "lab_catalog"
  },
  "endpoints": [
    "GET /v1/{prefix}/namespaces",
    "POST /v1/{prefix}/namespaces/{namespace}/tables/{table}",
    "POST /v1/{prefix}/transactions/commit",
    "GET /polaris/v1/{prefix}/applicable-policies"
  ]
}
```

endpoints は 36 個宣言されており、Iceberg 仕様のエンドポイントに加えて Polaris 固有の `polaris/v1/...`（generic-tables、policies）が混在しています。仕様は endpoints を文字列として扱うため、実装固有のエンドポイントもこの形で宣言できます。

`idempotency-key-lifetime` は返りませんでした。仕様の「無ければ非対応とみなせ」という規定により、**このサーバでは `Idempotency-Key` による安全なリトライが使えない**と判断します。500 を受け取ったときは、クライアント側でテーブルを読み直して自分のコミットが入っているか確認する必要があります。

`overrides.prefix` として `lab_catalog` が配られ、以降のベース URL は次になりました。

```
http://localhost:8181/api/catalog/v1/lab_catalog
```

このように prefix でカタログを分ける実装は一般的ですが、仕様は prefix の意味論を定義していません（description は “An optional prefix in the path” のみ）。解釈はサーバ実装に委ねられます。

namespace 作成は 409 を返しました。これは AlreadyExistsException であり、リトライ不可の 409 です。

```
--- POST /v1/{prefix}/namespaces — namespace 作成
    409: 既に存在します (AlreadyExistsException)
```

10.6.0.1 loadTable が返すもの

loadTable のレスポンスには、**テーブルの状態そのもの**が入っています。ポインタ（metadata-location）だけでなく、metadata.json の中身（metadata）も展開されて返るため、クライアントは S3 から metadata.json を読み直さずに済みます。

さらに storage-credentials が付くことがあります。これが **credential vending** です。

普通に考えると、クライアントが S3 のデータを読むには S3 の資格情報が必要です。しかしそれを全クライアントに配ると、**テーブル単位の権限管理ができません**。カタログ側で「このユーザはこのテーブルだけ」と決めても、S3 の鍵を持っていれば直接バケット全体を触れてしまうからです。

credential vending は、この矛盾を解きます。クライアントは S3 の鍵を持たず、カタログにテーブルを要求します。カタログは権限を確認したうえで、**そのテーブルのパスにだけ有効な、期限付きの鍵**を発行します。

今回のレスポンスは次のとおりでした。トップレベルキーは 4 つで、credential vending が有効でした。

```
    レスポンスのトップレベルキー: ['metadata-location', 'metadata', 'config', 'storage-
credentials']

    storage-credentials が 1 個返りました:
      prefix: s3://warehouse/lab_catalog/lab/rest_demo
        s3.access-key-id = ***（本文では伏せています）
        s3.secret-access-key = ***
        s3.session-token = ***
        expiration-time = 1784386178000
        s3.session-token-expires-at-ms = ***
```

注目すべきは prefix です。払い出された資格情報は**テーブル 1 つ分のパスにスコープされています**。カタログ全体でも namespace 単位でもありません。クライアントはこのテーブル配下しか触れない鍵を受け取ります。

s3.session-token が付いていることから、静的な鍵ではなく一時的な資格情報だと分かります。expiration-time は有効期限で、1970-01-01 からの経過ミリ秒（エポックミリ秒）で表されています。

storage-credentials は prefix 付きで返ります。クライアントは config より先に storage-credentials を参照し、複数返ってきた場合は、対象パスに最も長く一致するものを使います（longest-prefix-wins）。カタログ全体向けの鍵とテーブル個別の鍵が両方返るような場合に、より限定された後者が選ばれるということです。

嘘の snapshot ID を表明した updateTable のリクエストとレスポンスは次のとおりです。

```
{
  "identifier": { "namespace": ["lab"], "name": "rest_demo" },
  "requirements": [
    { "type": "assert-table-uuid", "uuid": "260da375-3cb1-4ea7-825e-911487d00cf7" },
    { "type": "assert-ref-snapshot-id", "ref": "main", "snapshot-id": 1234567890123456789 }
  ],
  "updates": [
    { "action": "set-properties", "updates": { "lab.marker": "should-not-apply" } }
```

```
  ]
}
```

```
    ステータス: 409

    レスポンス:
      {
        "error": {
          "message": "Requirement failed: branch main has changed: expected id 1234567890123456789 != 14724
          "type": "CommitFailedException",
          "code": 409
        }
      }
```

エラーメッセージが、期待値と実際の値の両方を含んでいる点に注目してください。実際の current-snapshot-id は 1472481652836649450 で、これは loadTable が返した値と一致します。updates 側の set-properties は適用されていません。requirements が 1 つでも失敗すれば updates は一切適用されません。一部だけ適用されて中途半端な状態が残ることはない、ということです。

requirements を正しい値に直すと成功し、metadata-location が差し替わりました。

```
    正しい requirements でコミットします:
    ステータス: 200
    新しい metadata-location: s3://warehouse/lab_catalog/lab/rest_demo/metadata/00002-02cf940b-
6a03-4c23-850d-3776e25e8efc.metadata.json
```

loadTable 時点では 00001-f856842a-...metadata.json でしたので、コミットによって 00002-... へ移ったことになります。コミット成功時のレスポンスには metadata-location と metadata が必ず含まれます（仕様上 required）。クライアントは差し替え後の状態を、読み直さずにそのまま受け取れます。

存在しないテーブルへの loadTable は 404 を返しました。

```
{
  "error": {
    "message": "Table does not exist: lab.does_not_exist",
    "type": "NoSuchTableException",
    "code": 404
  }
}
```

非 2xx はすべて {"error": {message, type, code}} という形でラップされます。中身（message / type / code）を error で包まずそのまま返す実装は仕様違反です。クライアントは type を見て処理を分けるため、この形が保たれていることが前提になります。

10.7 ここから分かること

コミットの実体は、requirements で読んだ版を表明し、サーバ側でその版が変わっていないことを確認したうえで、metadata.json のポインタを差し替える操作です。メタデータの構造そのものは 15. メタデータ三層構造を覗く で見たとおりで、REST カタログはその最上位のポインタだけを管理します。

クライアントを自作する場合、409 に対する CAS リトライループは必須です。テーブルを再ロードして最新の snapshot ID を取り、requirements を作り直して再送します。一方で 500 CommitStateUnknownException を 409 と同じ扱いにしてはいけません。結果が不明な状態でリトライすると、同じスナップショットが二重に適用されます。この実験で使ったサーバは `idempotency-key-lifetime` を返していないため、`Idempotency-Key` による保護は期待できません。500 を受けたら、再ロードして自分のコミットが適用済みかどうかを確認する経路が必要になります。

409 の二義性も実装上の注意点です。ステータスコードだけで分岐すると、AlreadyExistsException に対して無限にリトライする実装になりかねません。エラーボディの `type` を見て判断します。

`GET /v1/config` を最初に呼ぶ理由もここで具体化しました。prefix が overrides で配られる以上、config を呼ばずに URL を組み立てると全リクエストがパスを外します。endpoints と `idempotency-key-lifetime` も同様に、config を読まなければサーバの能力を判定できません。

REST 仕様側の整理は姉妹リポジトリの調査報告書（https://dobachi.github.io/iceberg-research/ ）にまとめてあります。

11 17. PyIceberg の制約を実証する

これまでのサンプルは「動くこと」を確認してきました。samples/07_pitfalls.py は逆に「動かないこと」を確認します。PyIceberg でできないことを実際に踏んで、エラーメッセージや戻り値として記録するのが目的です。

対象は PyIceberg 0.11.1、確認日は 2026-07-17（2026-07-18 に再確認）です。ここで挙げる項目は PyIceberg の欠陥ではなく、実装の進行状況です。Iceberg の仕様は Java 実装が先行しており、Python 実装は追随している途中にあります。ただし、知らずに設計すると後から構成を組み直すことになるため、事前に把握しておく必要があります。

11.1 確かめたいこと

- format-version 3 へのアップグレードが通らないこと
- `write.delete.mode = merge-on-read` を設定しても copy-on-write で実行されること
- equality delete が書けないだけでなく読めないこと
- メンテナンス機能に compaction / orphan file 削除が無いこと
- `s3.path-style-access` が PyIceberg に存在せず、黙って無視されること
- `upsert` が SQL の MERGE INTO と同等ではないこと

11.2 背景

Iceberg のテーブル仕様は format-version で世代管理されています。v2 で position delete / equality delete による merge-on-read が入り、v3 で deletion vector と row lineage が加わりました。PyIceberg は v3 のメタデータ型（`TableMetadataV3`）を持っており読み取りは可能ですが、書き込みは制限されています。追跡 issue は apache/iceberg-python#1551 です。

merge-on-read（MoR）と copy-on-write（CoW）は更新の実装方式の違いです。CoW は対象行を含むデータファイルを丸ごと書き直します。MoR は delete ファイルを追加するだけで、読み取り時に適用します。書き込みが軽い代わりに読み取りが重くなり、compaction が前提になります。

equality delete は「この列がこの値の行を消す」という形式の delete ファイルです。Flink の UPSERT（`write.upsert.enabled`）はこの形式を使います。

11.3 処理の流れ

1. PyIceberg のバージョンを表示し、テーブル `lab.pitfalls` を作って 5 行 append します。
2. `upgrade_table_version(3)` を試みます。

```
with tbl.transaction() as tx:
    tx.upgrade_table_version(3)
```

3. write.delete.mode = merge-on-read を設定してから delete を実行し、warnings.catch_warnings(record=True) で警告を捕捉します。そのうえで inspect.delete_files() の行数を確認します。
4. equality delete については、書けないため実際には作れません。PyIceberg のソースにある 2 つの raise を提示して代替します。1 つは書き込み経路の ValueError、もう 1 つは読み取り（REST scan planning）経路の NotImplementedError です。
5. tbl.maintenance の公開メソッドを列挙し、compaction / orphan 系が無いことを確認します。
6. pyiceberg.io に s3.path-style-access に相当する定数が定義されていないことを、モジュールの大文字定数を走査して確認します。

```
has_path_style = any(
    getattr(io_mod, n, None) == "s3.path-style-access"
    for n in dir(io_mod) if n.isupper()
)
```

7. tbl.upsert のシグネチャを表示し、実際に upsert を 1 回実行します。

11.4 実測結果

11.4.1 1. format-version 3

```
    PyIceberg バージョン: 0.11.1

---【落とし穴 1】format-version 3 は読めるが書けない
    現在の format-version: 2
    → 既定は 2 です (TableProperties.DEFAULT_FORMAT_VERSION = 2)

    v3 にアップグレードしてみます:
    OK  期待どおり拒否されました: ValueError: Unsupported table format version: 3
```

このサンプルは当初 NotImplementedError を想定して書かれていましたが、実際にはその手前で ValueError になりました。0.11.1 のソースを追うと、v3 は二重にガードされています。

順	場所	例外
1	table/__init__.py L338 の upgrade_table_version（format_version not in {1, 2}）	ValueError: Unsupported table format version: 3
1’	table/update/__init__.py L316（SUPPORTED_TABLE_FORMAT_VERSION = 2 と比較）	同上

順	場所	例外
2	table/metadata.py L578 の TableMetadataV3.model_dump_json	NotImplementedError: Writing V3 is not yet supported

手前の 1 で弾かれるため、2 には通常到達しません。到達する例外の種類は想定と異なりましたが、v3 へアップグレードできないという結論は変わりません。

型定義を見ると typedef.py L212 の TableVersion = Literal[1, 2, 3] は v3 を許容しているように読めます。しかし実際に書けるのは v2 までです。**型定義だけを見て対応状況を判断すると誤ります。**

実務上は、PyIceberg では deletion vector も row lineage も使えず、v3 を前提とした設計は現時点で成り立ちません。

11.4.2 2. merge-on-read

プロパティの設定自体は成功します。しかし delete の実行時に警告が出て、copy-on-write で処理されます。

```
テーブルプロパティを設定しました: write.delete.mode = merge-on-read

この状態で delete を実行します:
OK  警告が出ました: Merge on read is not yet supported, falling back to copy-on-write
OK  delete ファイル数: 0 (CoW なので 0 が期待値)
```

inspect.delete_files() が 0 行であることから、position delete が 1 つも書かれていないと確認できます。プロパティは受け付けられて tbl.properties にも残るため、テーブル定義だけを見ると MoR で運用しているように見えます。実際の挙動との差は警告でしか現れません。

11.4.3 3. equality delete

書き込みができないため、このサンプルでは実データを作れていません。ソース上の raise で示しています。

```
ソース (table/__init__.py L2114):
  raise ValueError("PyIceberg does not yet support equality deletes: ...")

ソース (table/__init__.py L1886, REST scan planning 経路):
  raise NotImplementedError(f"PyIceberg does not yet support equality deletes: {delete_file.file_path}'
```

上の 2 つはそれぞれ書き込み経路 L2114 と読み取り経路 L1886 にあります。

後者が読み取りパスにあることが実務上の要点です。Flink や Spark が equality delete を書いたテーブルは、PyIceberg では開けません。この経路自体はラボ内で実行できていないため、実際の失敗を再現した結果としては未確認です。上流の書き込みエンジンが equality delete を使うかどうかは、PyIceberg を読み取り側に据える前に確認する必要があります。

11.4.4 4. メンテナンス機能

```
MaintenanceTable のメソッド: ['expire_snapshots', 'tbl']

OK  expire_snapshots は存在する
OK  compaction / rewrite_data_files は存在しない
OK  remove_orphan_files は存在しない
```

公開されているのは `expire_snapshots` だけでした。実行自体は通ります。

```
expire_snapshots は使えます:
  スナップショット数: 2 -> 2
  (10 年前より古いものを対象にしたので、何も消えません)
```

compaction が無いことは、15. メタデータ三層構造を覗く で見た小ファイル問題に直結します。あの実験では manifest ファイルのほうが data file より大きいという状態が発生していました。これを解消する手段が PyIceberg 側にありません。

11.4.5 5. s3.path-style-access

```
実在するキー: S3_ENDPOINT = 's3.endpoint'
実在するキー: S3_FORCE_VIRTUAL_ADDRESSING = 's3.force-virtual-addressing'
OK  s3.path-style-access は定義されていない
```

`s3.path-style-access` は Java 版 Iceberg / Spark のキーです。PyIceberg の設定に書いてもエラーにはならず、未知のキーとして無視されます。プロパティが単なる dict 参照であるため、設定ミスが表面化しません。

一方で、MinIO や RustFS のような S3 互換ストレージに対しては、何も設定しなくても path-style になります。PyArrow の `S3FileSystem` は `force_virtual_addressing` の既定が False で、公式ドキュメントは次のように述べています。

> If false, then virtual addressing is only enabled if endpoint_override is empty

`s3.endpoint` を設定した時点で `endpoint_override` が非空になるため、自動的に path-style が選ばれます。

混乱しやすいのは、同じ compose ファイルの中に 2 つの流儀が同居する点です。Java 側のコンテナには `CATALOG_S3_PATH__STYLE__ACCESS=true` が効き、Python 側には対応するキーが存在しません。

11.4.6 6. upsert

```
upsert のシグネチャ:
  df = (必須)
  join_cols = None
  when_matched_update_all = True
  when_not_matched_insert_all = True
  case_sensitive = True
```

```
      branch = main
      snapshot_properties = {}
```

分岐の制御は `when_matched_update_all` と `when_not_matched_insert_all` の真偽値 2 つだけです。SQL の MERGE INTO にある「WHEN MATCHED AND 条件 THEN UPDATE SET 特定列」「WHEN MATCHED THEN DELETE」「複数の WHEN 句」は表現できません。

実行自体は期待どおり動きます。

```
      更新: 1 行, 挿入: 1 行
 id      val
  1      NEW
  2     keep
  3 inserted
```

id=1 が更新され、id=3 が挿入され、id=2 はそのまま残りました。ただし内部は delete ファイルを使わない CoW であり、対象を含むデータファイルが書き直されます。

11.5 ここから分かること

PyIceberg 0.11.1 は、読み取りと軽量な書き込み、メタデータの調査に向いています。`inspect.*` は充実しており、15. メタデータ三層構造を覗く のような調査はこれだけで完結します。DuckDB / Polars / pandas との連携も同様です。REST カタログ仕様の学習と検証にも使えます。

一方で次の用途には現時点で適しません。

- format-version 3 の機能（deletion vector、row lineage）を使う設計
- Flink / Spark が equality delete を書くテーブルの読み取り
- merge-on-read での更新（CoW に落ちる）
- compaction や orphan file 削除を含む運用の自動化
- 複数条件を持つ MERGE INTO 相当のロジック

現実的な構成は、PyIceberg で読み取りと軽い書き込みを担い、メンテナンスと重い DML は Spark に任せる形になります。「Python だけで Iceberg を運用する」は、少なくとも 0.11.1 の時点では成立しません。ブランチ運用についても、14. タイムトラベルと branch / tag で触れたとおり fast-forward が Spark procedure であるため、同じ結論になります。

これらの状況は実装の進行に伴って変わります。バージョンを上げる際は、ここで確認した各項目を再度実行して現況を確かめてください。運用面の背景は姉妹リポジトリの調査報告書（https://dobachi.github.io/iceberg-research/ ）にまとめてあります。

12 90. 分かったこと

実験を通して確認できたことを集約します。各項目の詳細は、対応する実験のページを参照してください。

すべて **2026-07-17 時点**、**Apache Iceberg 1.11.0 / PyIceberg 0.11.1 / Apache Polaris 1.6.0** での観測です。

12.1 裏が取れたこと

文献で読んだ内容を、実際に動かして確認できたものです。

主張	観測結果	詳細
新規テーブルの既定は format-version 2	`format_version = 2` を確認	11
削除の既定は copy-on-write	delete 後に delete ファイルが 0 件、データファイルが書き直された	11 17
rename しても field ID は変わらない	rename 前後で field ID が一致	12
パーティション進化は既存データを書き換えない	新旧 2 つの spec_id がファイル単位で共存	13
hidden partitioning はパーティション列を書かせない	論理列 `ts` の述語だけで正しく絞り込めた（枝刈りの効き自体は本規模では非観測）	13
タグはスナップショットの寿命を延ばす	参照されたスナップショットが保持される	14
コミットはポインタのアトミックな差し替え	`metadata-location` が新ファイルに切り替わる	15 16
credential vending が動作する	テーブル単位にスコープされた短命の資格情報が払い出された	16
requirements 不一致は 409 になる	意図的に古い snapshot-id を表明して 409 を再現	16
PyIceberg は v3 を書けない	`ValueError: Unsupported table format version: 3` で拒否（書き込み直前の `NotImplementedError` は手前のガードのため到達せず）	17
merge-on-read は copy-on-write に落ちる	警告が出て delete ファイルが作られない	17

12.2 動かして初めて分かったこと

文献からは読み取れず、実際に構築する過程で判明したものです。

12.2.1 S3 互換ストレージでは roleArn を設定してはいけない

カタログの storage config に `roleArn` を書くと、コミット時に次で失敗します。

```
pyiceberg.exceptions.CommitStateUnknownException:
  StsException: (Service: Sts, Status Code: 400, Request ID: null)
```

`roleArn` があると Polaris は credential vending のために STS AssumeRole を呼びますが、RustFS も MinIO も STS に対応していません。省略すれば静的な資格情報が使われ、credential vending も機能します。

このエラーには紛らわしさがあります。`CommitStateUnknownException` は仕様上「コミットが成功したか失敗したか分からない」という重大な状態を指しますが、この場合の実態は単なる設定ミスです。**エラーの型名から深刻度を判断すると誤ります。**

12.2.2 s3.path-style-access は PyIceberg に存在せず、黙って無視される

このキーは Java 版 Iceberg / Spark のものです。PyIceberg に書いてもエラーにならず無視されます。プロパティが単なる dict 参照のため、未知のキーが検出されないからです。

さらに、path-style を使いたいなら**何も設定しなくて構いません**。PyArrow の `S3FileSystem` は `force_virtual_addressing` の既定が `False` で、`endpoint_override` が設定されていれば path-style になります。つまり `s3.endpoint` を設定した時点で目的は達成されています。

同じ compose の中で Java 側は `CATALOG_S3_PATH__STYLE__ACCESS=true` が有効、Python 側は同名概念のキーが存在しない、という非対称が生じます。ここは混乱しやすい箇所です。

12.2.3 スコープでロールを絞っても、権限は絞られなかった

Polaris は OAuth2 の `scope` を「どの principal role を有効化するか」の指定として使います。`PRINCIPAL_ROLE:ALL` が全ロール、`PRINCIPAL_ROLE:<名前>` が個別指定です。

実装（`polaris-runtime-service-1.6.0.jar` を逆アセンブルして確認）では、トークン発行時に書式だけを検証し、ロールの実在は確認しません。認証時にはロール名で principal のロールを絞り込み、見つからなければ警告を記録して空集合のまま続行します。

問題は、**有効ロールが空のトークンでもテーブルを新規作成できた**ことです。ログには `roles=[]` と警告が出ているにもかかわらず、`createTable` は HTTP 200 を返しました。`purgeRequested=true` 付きの削除は 403 になったので、認可が完全に無効というわけではありません。

スコープを権限の絞り込み手段として当てにしないほうが安全です。なぜ空のロール集合で書き込みが通るのかは追えていません（未確認）。なお確認したのは、仕様上 DEPRECATED for REMOVAL とされている `/v1/oauth/tokens` 経由の挙動です。推奨される外部 IdP 構成での挙動は確認していません。詳細は 01. 環境構成 を参照してください。

12.2.4 型名が実態を表さないことがある

上の `CommitStateUnknownException` に加えて、409 にも 2 つの意味があります。

状況	意味	リトライ
namespace 作成での 409	既に存在する	不可（リトライしても同じ）
updateTable での 409	コミット競合	可能（再ロードして再コミット）

同じステータスコードでも対処が正反対です。詳しくは 16. REST API を直接叩く を参照してください。

12.3 実装の成熟度について確認できたこと

PyIceberg 0.11.1 で「できないこと」を実際に踏んで確認しました。これらは欠陥ではなく成熟度の問題ですが、知らずに設計すると行き詰まります。

できないこと	実態
format-version 3 の書き込み	型は実在しパースは可能。アップグレードは `ValueError` で拒否される（`NotImplementedError` のガードもあるが手前で弾かれ到達しない）。**deletion vector も row lineage も使えません**
equality delete の読み取り	書けないだけでなく**読めません**。Flink の UPSERT は equality delete ベースなので、その組み合わせは実害が出ます
merge-on-read での書き込み	プロパティを設定しても警告を出して copy-on-write に落ちます
compaction / orphan file 削除	`MaintenanceTable` は `expire_snapshots` のみ。小ファイル問題を PyIceberg 単体で解決できません
MERGE INTO 相当	`upsert()` は真偽値 2 つのみ。条件付き句も `WHEN MATCHED THEN DELETE` もありません

ここから導かれる現実的な構成は、**PyIceberg で読み取りと軽い書き込み、メンテナンスと重い DML は Spark に任せる**というものです。「Python だけで Iceberg を運用する」は現時点で成立しません。

12.4 設計上、注意が必要だと分かったこと

12.4.1 コミット 1 回がメタデータ 1 世代を生む

append を 1 回行うたびにスナップショットが 1 つ、metadata JSON が 1 つ増えます。これは最適化の余地ではなく、**原子性を実現するための構造的な要件**です。高頻度コミットがメタデータ肥大を招くのは、この構造の直接の帰結です。

あわせて既定値に注意が要ります。

- `write.metadata.previous-versions-max` の既定は 100
- `write.metadata.delete-after-commit.enabled` の既定は **false**

つまり何もしないと metadata JSON は溜まり続けます。しかも 100 個を溢れた分は追跡対象から外れ、後から削除を有効化しても消せません。

12.4.2 タグを 1 つ打つと自動メンテナンスが止まることがある

タグやブランチから参照されているスナップショットは削除されません。これは仕様どおりの挙動です。

問題は、マネージドサービスでの扱いです。Amazon S3 Tables の公式ドキュメントによれば、ユーザー定義のタグやブランチが存在すると snapshot management がテーブル全体で失敗します。**タグを 1 つ打っただけで自動メンテナンスが全面停止し、それが課金の増加としてしか現れません。**

12.4.3 統計の既定値がプルーニングを効かなくすることがある

- `write.metadata.metrics.default = truncate(16)` --- string の境界値は 16 文字で切り詰められます。URL や UUID のように共通プレフィックスが長い列では境界値が実質同一になり、file skipping が効きません。
- `write.metadata.metrics.max-inferred-column-defaults = 100` --- 101 列目以降には既定でメトリクスが収集されません。ワイドテーブルで後方の列に述語をかけると全ファイル読みになります。

どちらも「遅い」という形でしか表面化せず、原因にたどり着きにくい部類です。

12.5 このラボで確かめられなかったこと

範囲外だったものを明示しておきます。

対象	理由
性能・スケール特性	ローカル単一ノードでは意味のある測定ができません
compaction の実際	Spark procedure が必要で、本ラボの構成では試せません
v3 機能（deletion vector / row lineage）	PyIceberg が書けないため。Spark を繋げば検証できます
equality delete を含むテーブルの読み取り	書き手（Flink 等）が用意できないため未検証
複数エンジンからの同時コミット競合	409 は API を直接叩いて再現しましたが、実エンジン同士の競合は未検証
本番相当の認証（外部 IdP）	`/v1/oauth/tokens` を使用。これは仕様上、削除予定のエンドポイントです

Spark を繋げば上の多くが検証できます。Polaris 公式の guides/spark が出発点になります。

12.6 総括

このラボで得られた実感は、**Iceberg は仕様と実装の距離が大きい**ということです。

仕様は v3 で deletion vector を定義し、v4 が議論されています。一方 PyIceberg は v3 を書けず、Java 実装は仕様未採択の v4 を既に受け付けます。「Iceberg が対応している」という表現は、**どの仕様バージョンの、どの実装の話か**を伴わなければ意味を持ちません。

同時に、中核の設計は確かめた範囲では一貫していました。field ID による解決、manifest を書いた時点の spec で解釈する規則、ポインタ差し替えによるコミット --- これらは互いに噛み合っており、スキーマ進化とパーティション進化が既存データを書き換えずに成立する理由になっています。ここは文献の説明どおりでした。

判断に使う際は、調査報告書とあわせて、必ず一次情報で裏を取ってください。