databricks

ストリーミングを シンプルに！ Delta Live Tables

Akihiro Kuwano

Solutions Architect

スピーカー

Akihiro Kuwano / 桑野 章弘

データブリックス・ジャパン株式会社

Solutions Architect

経歴

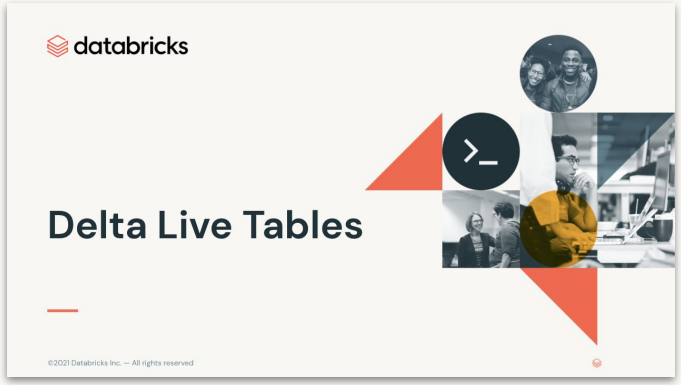
- B2C企業でのインフラエンジニアとしてのキャリアや、パブリッククラウドベンダーでソリューションアーキテクトとしてのキャリアを重ねる
- DatabricksでもB2C企業担当のソリューションアーキテクトとして様々な案件において技術支援を実施

はじめに

- 本日のセッションは後日オンデマンドでもご覧いただけます。
- セッション終了後に表示されるアンケートにご回答いただけますと、本日の資料を共有いたします。
- ご質問がありましたら、Q&Aボックスにご入力ください。後日回答します。
- 今後より良いコンテンツをお送りするためにも、ぜひアンケートへのご協力をよろしくお願いいたします。

詳細はこちら： go/dlt

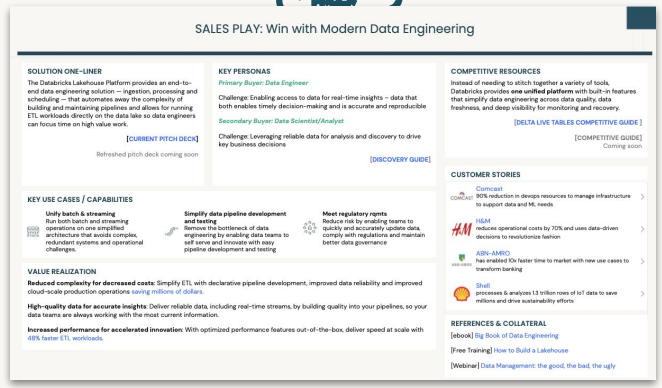
概要スライド



ユーザーガイド

セールス・プレイ：

最新のデータエンジニアリングで勝つ

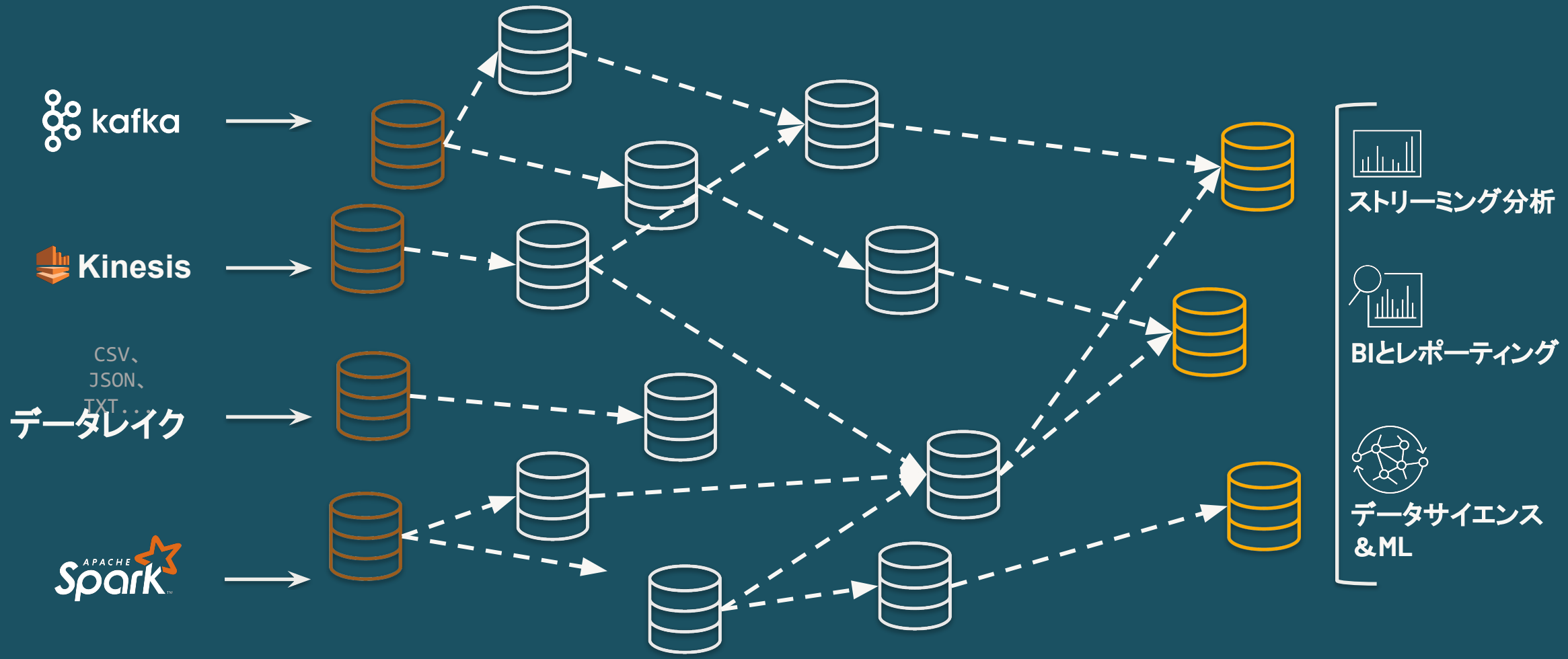


良いデータはレイクハウスの基礎

全データプロフェッショナルは、クリーンで新鮮そして信頼できるデータを必要としている

しかし、現実はそれほど単純ではない。

データ品質と信頼性を大規模に維持することは、しばしば、複雑で大変



データ・プロフェッショナルとしての人生...

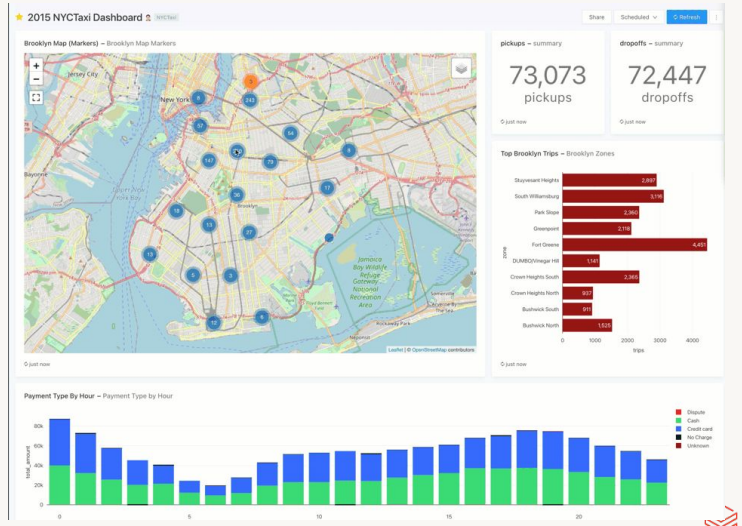
From: The CEO
Subject : 至急分析してくれ！
To: マイケル・アームブラスト

マイケル、こんにちは。過去数四半期にわたる純顧客維持率とその変化について、簡単な分析が必要です。生データは s3://our-data-bucket/raw_data/... にあります。

```
1 %fs ls /data/sensors
```

	path
1	dbfs:/data/sensors/_SUCCESS
2	dbfs:/data/sensors/_committed_3908896360792309052
3	dbfs:/data/sensors/_started_3908896360792309052
4	dbfs:/data/sensors/part-00000-tid-3908896360792309052-adec30c0-9ba8-4344-a36c-7ec3
5	dbfs:/data/sensors/part-00001-tid-3908896360792309052-adec30c0-9ba8-4344-a36c-7ec3
6	dbfs:/data/sensors/part-00002-tid-3908896360792309052-adec30c0-9ba8-4344-a36c-7ec3
7	dbfs:/data/sensors/part-00003-tid-3908896360792309052-adec30c0-9ba8-4344-a36c-7ec3

???



クエリから本番環境への移行

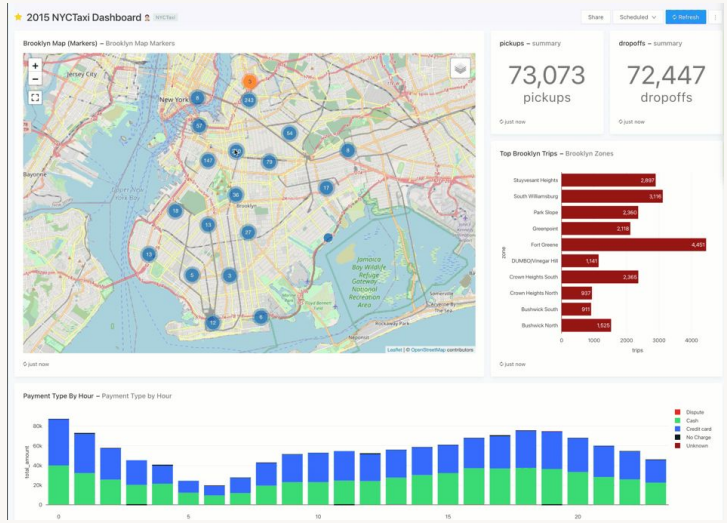
SQLクエリを信頼できるETLパイプラインに変換するために必要な面倒な作業

From： The CEO <ali@databricks.com>
Subject： 至急分析してくれ！
To： マイケル・アームブラスト<michael@databrick.com>

毎分
素晴らしいレポートだ！ありがとう~~毎日~~更新してくれる？

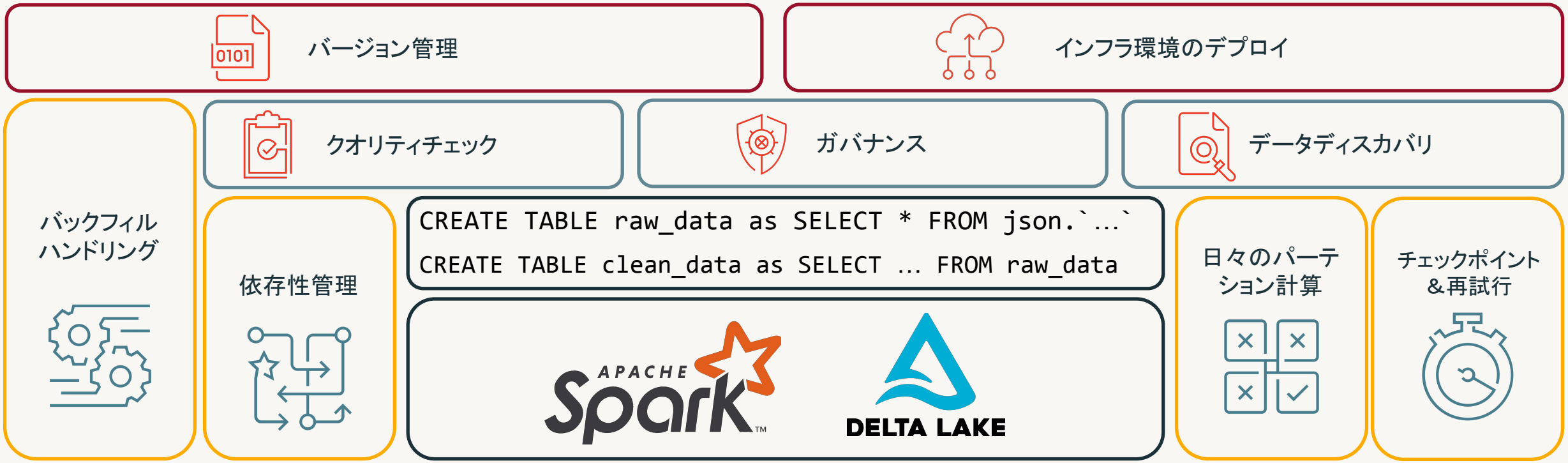
```
CREATE TABLE raw_data as SELECT * FROM json.`...`.
CREATE TABLE clean_data as SELECT ... FROM raw_data
```

APACHE Spark™ DELTA LAKE



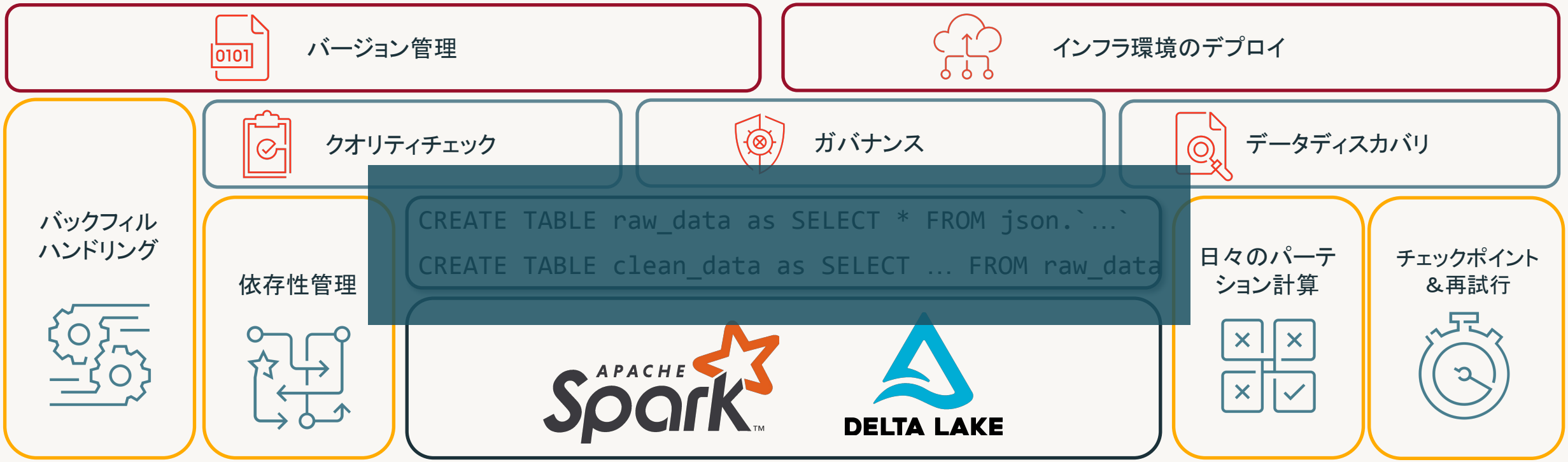
クエリから本番環境までの苦労

SQLクエリを信頼性の高いETLパイプラインに変換するために必要な退屈な作業



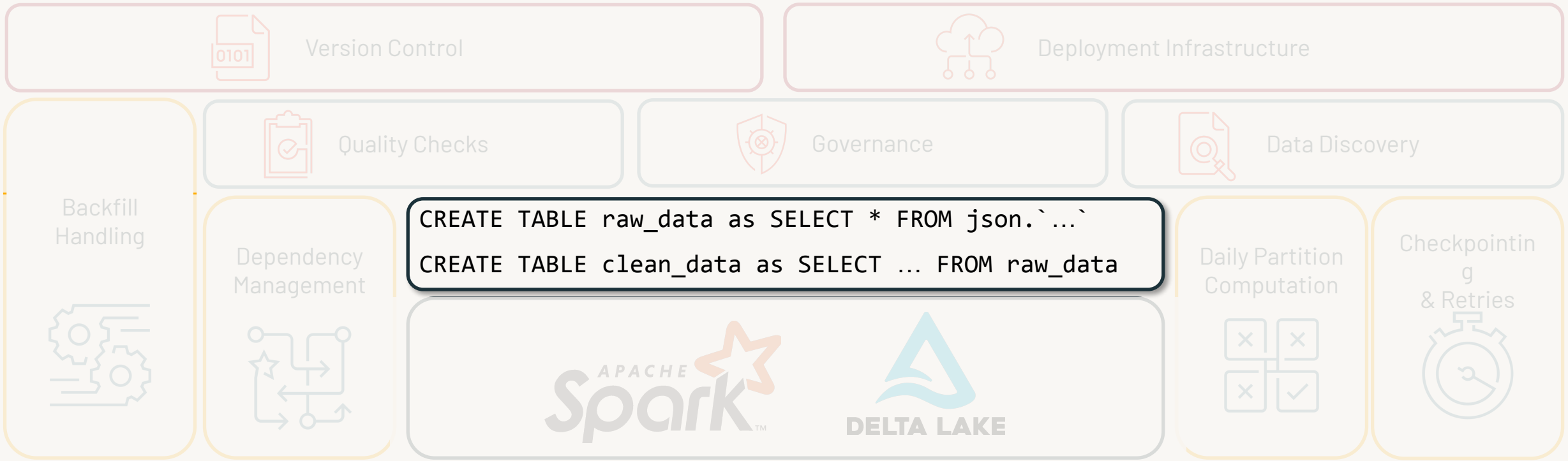
運用の複雑さが支配的

時間が**変換**ではなく***ツール作り***に費やされている



どこに時間を集中させるべき？

データから価値を引き出す



Delta Live Tablesの紹介

DLTを使用するだけで、クエリから本番パイプラインまでを一気通貫に処理が可能

```
CREATE STREAMING TABLE raw_data as SELECT * FROM cloud_files(…)
CREATE MATERIALIZED VIEW clean_data as SELECT … FROM LIVE.raw_data
```

Repos

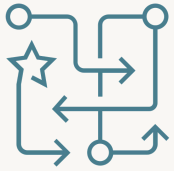
Unity Catalog

Databricks Workflows

Delta Live Tables

フル更新

依存性管理



Expectations

インクリメンタル計算

チェックポイント&再試行

宣言型プログラミング入門

宣言型プログラムは、「 何をすべきか 」を述べるものであり、「 それをどのように行うか 」ではない

命令型プログラム

```
numbers = […]
sum = 0
for n in numbers:
  sum += n
print(n)
```

宣言型プログラム

```
SELECT sum(n) FROM numbers
```

クエリオプティマイザが合計を計算する最適な方法を見つけ出す

データ独立性 物理的な変更（パーティショニング、ストレージの場所、インデックスなど）があってもクエリを書き直す必要がない

DLTを用いた宣言型プログラミング

宣言型プログラムは、「 何をすべきか 」を述べるものであり、「 それをどのように行うか 」ではない

Spark 命令型プログラム

```
date = current_date()
spark.read.table(“orders”)
  .where(s"date = $date")
  .select("sum(sales)")
  .write
  .mode(“overwrite”)
  .replaceWhere(s"date = $date")
  .table("sales")
```

DLT 宣言型プログラム

```
CREATE MATERIALZED VIEW sales
AS SELECT date, sum(sales)
FROM orders
GROUP BY date
```

DLTランタイムがこのテーブルを作成または更新する最適な方法を見つけ出す

ワークフロー それとも DLT？

多くの場合両方 ： ワークフローは DLTを含む 何でもオーケストレーション可

ワークフロー を使用し、どんなタスクでも実行

- スケジュール
- 他のタスクが完了した後
- ファイルが届いたとき
- 別のテーブルが更新されたとき

データフロー管理にはDLT を使用

- デルタテーブルの作成/更新
- 構造化ストリーミングの実行

DLT の中核となる抽象化

データセットを定義すれば、DLTが自動的にそれらを最新の状態に保つ

ストリーミング・テーブル

デルタ・テーブルにストリーム が書き込まれる

以下の用途に使用される:
- インジェスト
- 低レイテンシーの変換
- 巨大なスケール

マテリアライズド・ビュー

クエリの結果 、デルタテーブルに格納される

以下の用途に使用される:
- データの変換
- 集計テーブルの構築
- BIクエリとレポートの高速化

ストリーミング・テーブル とは？

Delta Tableに構造化ストリーミングが書き込まれる

```
CREATE STREAMING TABLE report
AS SELECT *
FROM cloud_files("/mydata/", "json")
```

- ストリーミング・テーブルKafka、Kinesis、AutoLoader（クラウド・ストレージ上のファイル）など、アペンド・オンリーのデータ・ソースから読み込む
- 各入力レコードを一度だけ読み込むことで、コストとレイテンシを削減できます
- ストリーミングテーブルはDML(UPDATE、DELETE、MERGE) をサポートし、アドホックなデータ操作 (GDPR など) を可能にします

マテリアライズド・ビュー とは？

クエリの結果 、事前に計算され、Deltaに保存される

```
CREATE MATERIALIZED VIEW report
AS SELECT sum(profit)
FROM prod.sales
GROUP BY date
```

- マテリアライズド・ビューは、、常に最後に更新された時点（つまり、スナップショット）で定義されたクエリ の結果を返します。
- マテリアライズド・ビューのデータを変更することはできませんが、そのクエリ を変更することはできます。

Expectationsを用いたデータ品質管理

Expectationsを使用した正確性の確保

Expectationsは本番環境でデータ品質を確保するためのテスト

```
CONSTRAINT valid_timestamp
EXPECT (timestamp > '2012-01-01')
ON VIOLATION DROP
```

```
@dlt.expect_or_drop(
  "valid_timestamp",
  col("timestamp") > '2012-01-01')
```

Expectations(期待値)とは、処理中に各行を検証するために使用される真偽式のことである。

DLTは期待に違反したレコードをどのように扱うか、柔軟なポリシーを提供:

- Track　不良レコードの数を追跡
- Drop　不良レコードを削除
- Abort　不良レコードに対して処理を中断

SQLの機能を活用したExpectations

SQLの集計機能と結合を使用して複雑な検証を行う

```
-- 主キーが常に一意であることを確認してください！
CREATE MATERIALIZED VIEW report_pk_tests(
  CONSTRAINT unique_pk EXPECT (num_entries = 1)
)
AS SELECT pk, count(*) as num_entries
FROM LIVE.report
GROUP BY pk
```

SQLの機能を活用したExpectations

SQLの集計機能と結合を使用して複雑な検証を行う

```
-- 二つのテーブル間でレコードを比較する
-- または、外部キー制約を検証 する
CREATE MATERIALIZED report_compare_tests(
  CONSTRAINT no_missing EXPECT (r.key IS NOT NULL)
)
AS SELECT * FROM LIVE.validation_copy v
LEFT OUTER JOIN LIVE.report r ON v.key = r.key
```

Announcing: DLT in DBSQL

PREVIEW

DLTはDBSQLで作成したストリーミングテーブルとマテリアライズド・ビューを強化

1つのSQLコマンドで堅牢でスケーラブルな
増分取り込みパイプラインを実現

ここでちょっとメモ：変化している用語

今までのDLTのユーザーは、名前が進化していることに気付くでしょう

- STREAMING LIVE TABLE ➡ STREAMING TABLE
- LIVE TABLE ➡ MATERIALIZED VIEW

意味は同じ、互換性のために古い構文もサポート

目標は構文を簡素化し、他のシステムと合わせること

Deep dive into Streaming

Spark™ Structured Streaming が基盤

ストリーミング・テーブルは、Spark Structured Streaming と Delta Table を組み合わせたもの

Streaming Computation Model(ストリーミング計算モデル):

入力は成長する追加専用テーブル

- クラウドストレージにアップロードされたファイル
- kafka、kinesis、eventhubのようなメッセージバス
- delta.appendOnly=trueのDelta Table
- 他のデータベースのトランザクションログ

すべてのデータが到着するまで待つのではなく、構造化ストリーミングは結果をオンデマンドで生成

- 更新のたびに処理するデータを少なくすることで、待ち時間を短縮する。
- 冗長な作業を避けることによるコスト削減

ストリーミング＝高価とは限らない

Delta Live Tablesでは、結果を更新する頻度を選択できる

トリガー: 手動

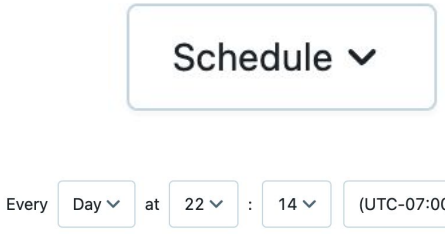
コスト 最低

待ち時間 最高

トリガー: スケジュールされた Databricks ジョブ

コスト 頻度による

待ち時間 10分～数ヶ月

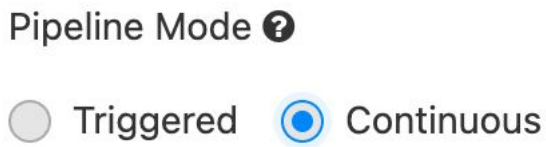


継続実行

コスト 最高

レイテンシー 分～秒

(ワークロードによっては)



取り込みにストリーミングテーブルを使用する

クラウドストレージにアップロードされたファイルを容易に取り込める

```
CREATE STREAMING TABLE raw_data
AS SELECT *
FROM cloud_files("/data", "json")
```

この例では、"/data "に格納されているすべてのjsonデータを含むテーブルを作成している：

- `cloud_files` どのファイルが読み込まれたかを追跡し、重複や無駄な作業を避けられる
- 任意のスケールでリスティングと通知の両方をサポート
- 設定可能なスキーマ推論とスキーマ進化

Spark™ Structured Streaming を使ったデータ取り込み

メッセージバスからレコードを簡単に取り込むことが可能

```
@dlt.table
def kafka_data():
  return spark.readStream \
    .option("format", "kafka") \
    .option("subscribe", "events") \
    .load()
```

この例では、Kafkaトピック「events」に公開されたすべてのレコードを含むテーブルを作成

- KafkaソースとDLTは、どのパーティション/オフセットが既に読み取られたかを自動的に追跡
- DBRに含まれる任意の構造化ストリーミングソースをDLTで使用可能
- メッセージバスはデータ取り込みを最も低レイテンシーに実行可能

SQL STREAM() ファンクションを使う

任意の Delta Table からデータをストリーミングする

```
CREATE STREAMING TABLE mystream
  AS SELECT …
  FROM STREAM(prod.events)
```

```
CREATE STREAMING TABLE mystream
  AS SELECT …
  FROM STREAM(delta.`s3:/…`)
```

- STREAM(. . .) は、新しく挿入されたレコードが到着するとテーブルを読み込む
- 追加のみ（Append Only）の Delta Table でのみ動作します

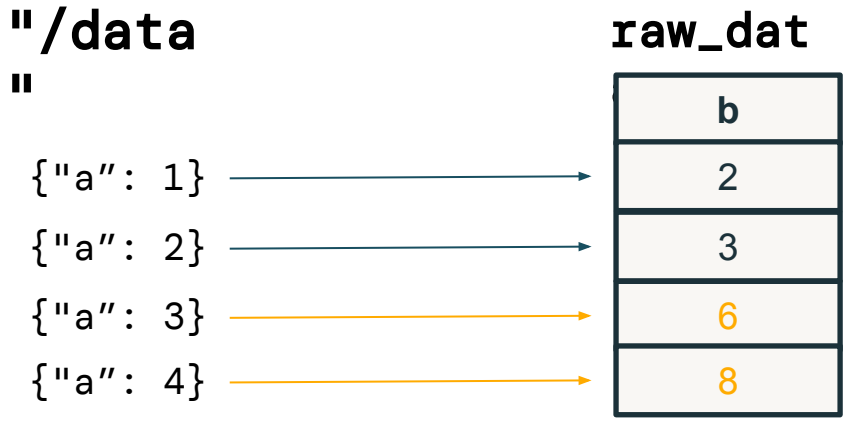
streaming ステートの理解

ストリーミング・テーブルは ステートフル

クエリが変更されても、各入力行は一度だけ処理されます

ストリーミング・ライブ・テーブルの定義に対する変更は、すでに処理されたデータを再読み込みすることはない：

```
CREATE STREAMING TABLE raw_data
AS SELECT a + 1 AS a    a * 2 AS a
FROM cloud_files("/data", "json")
```

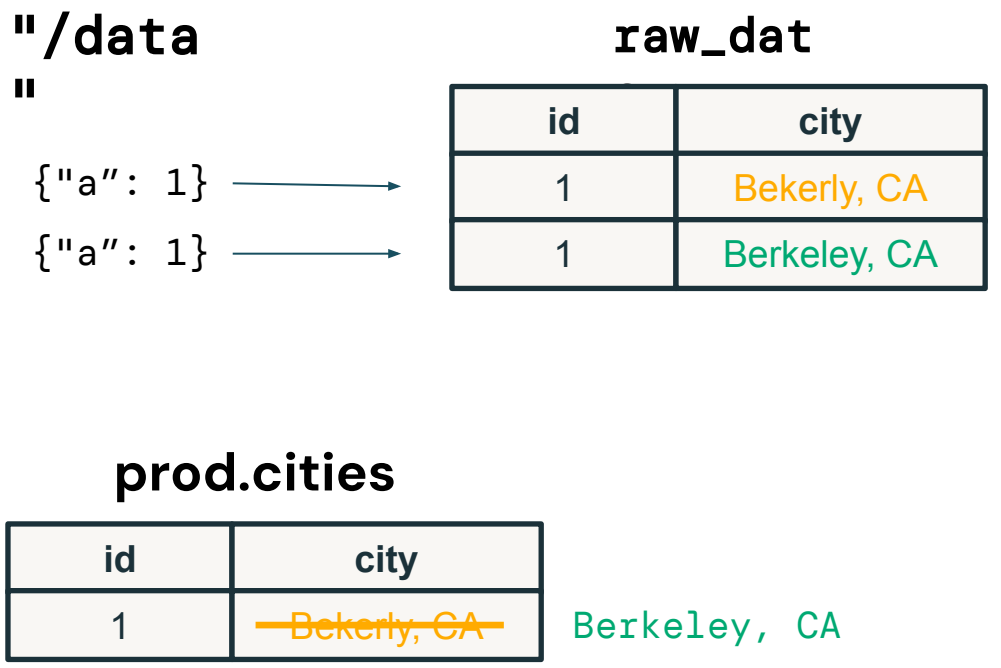


ストリーミング結合はステートフル

Deltaに格納された最新のスナップショットと結合しデータをエンリッチする

- スナップショットが変更された場合、マイクロバッチごとに自動的に更新される
- 結合されたテーブルのスナップショットに変更があっても、結果は再計算されない：

```
CREATE STREAMING TABLE raw_data
AS SELECT *
FROM cloud_files("/data", "json") f
JOIN prod.cities c USING id
```



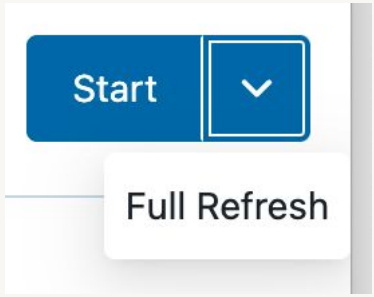
フルリフレッシュを使用してデータを再処理

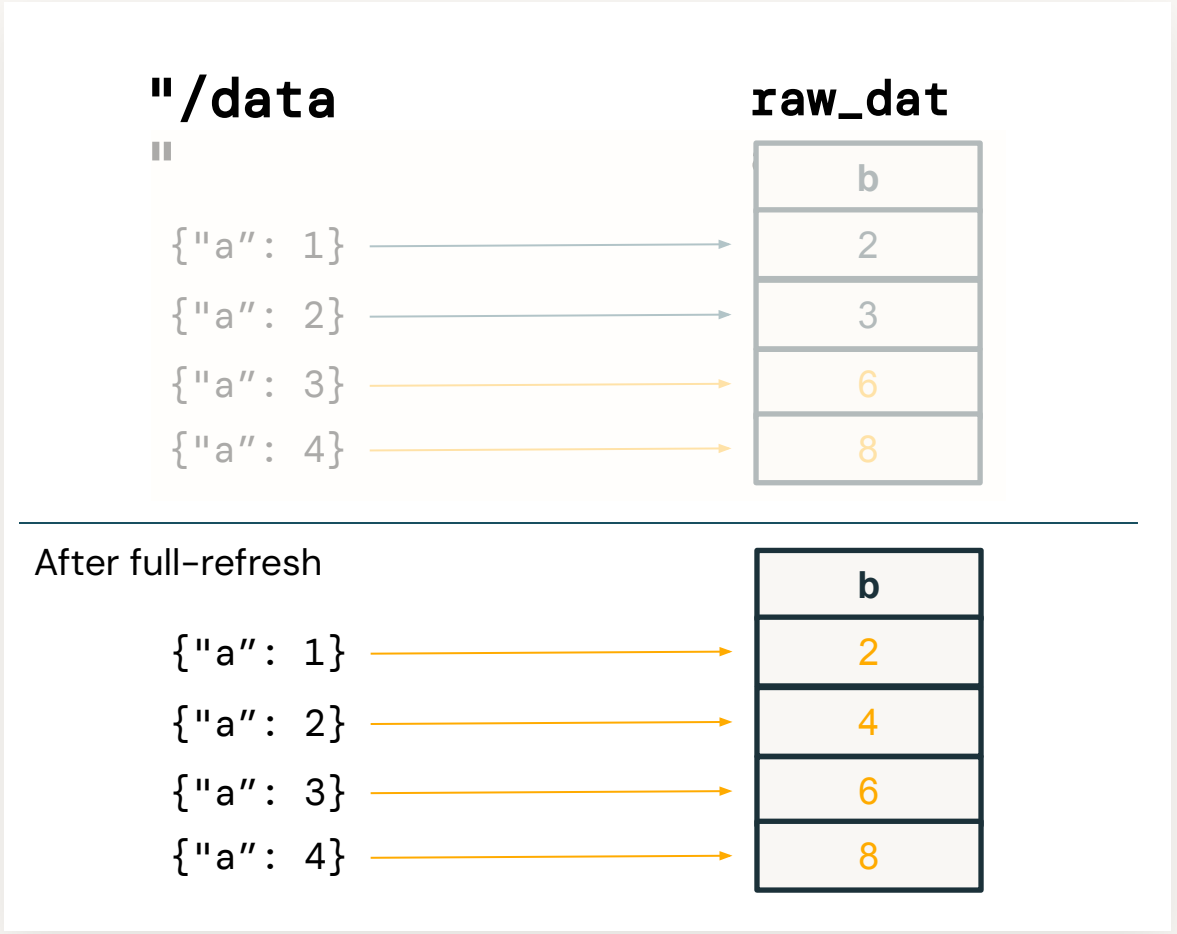
重要な変更後に完全リフレッシュを使用して自動的にバックフィルを実行

フルリフレッシュはテーブルのデータとクエリの状態を消去し、すべてのデータを再処理

```
CREATE STREAMING TABLE raw_data
AS SELECT a * 2 AS a
FROM cloud_files("/data", "json")
```

```
REFRESH raw_data FULL
```





stream-stream joins の使用

ストリーム-ストリーム結合によるファクト結合

ストリーム-ストリーム結合では互いに近接して到着したファクトを結合可能

```
@dlt.table()
def joined():
  impressions = read_stream("impressions") \
    .withWatermark("impressionTime", "10 minutes")

  clicks = clicks \
    .withWatermark("clickTime", "20 minutes")

  return impressions.join(clicks,
    expr("""clickAdId = impressionAdId AND
            clickTime >= impressionTime AND
            clickTime <= impressionTime + interval 5 minutes"""))
```

この例はアドテクを例にしていますが、クリックストリームと、その原因となった広告インプレッションに関する詳細を結合している

スナップショット-ストリーム結合とは異なり、ストリーム -ストリーム結合では、一致するレコードが現れるまで 、レコードをバッファリングする

ストリーム-ストリーム結合によるファクト結合

ストリーム-ストリーム結合では互いに近接して到着したファクトを結合可能

```
@dlt.table()
def joined():
  impressions = read_stream("impressions") \
    .withWatermark("impressionTime", "10 minutes")

  clicks = clicks \
    .withWatermark("clickTime", "20 minutes")

  return impressions.join(clicks,
    expr("""clickAdId = impressionAdId AND
            clickTime >= impressionTime AND
            clickTime <= impressionTime + interval 5 minutes"""))
```

2つのファクトのストリーム、それぞれにウォーターマークを持つ

結合条件
各ファクトがどれくらいの間隔で到着できるかという時間的な制約を記述

注意：ウォーターマークやタイムバウンドが欠落している場合、結合はすべてのデータを永遠にバッファリングする

ストリーミングソース
の追加と削除

ストリーミング・テーブルとフロー

DLT では、フローは新しいデータでテーブルを更新するオブジェクト

これまで示してきたシンプルな構文は、1つのDDLコマンドでテーブルとフローを作成するための糖衣構文である

```
CREATE STREAMING TABLE raw_data
AS SELECT * FROM kafka(…)
```

⟺

```
CREATE STREAMING TABLE raw_data

CREATE FLOW raw_data
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)
```

Checkpoints are identified by the name of the flow

```
checkpoints/raw_data
```

Kafka Offsets

```
{
  0: 123354,
  2: 9573943,
  3: 83275092…
}
```

マルチフロー・テーブル

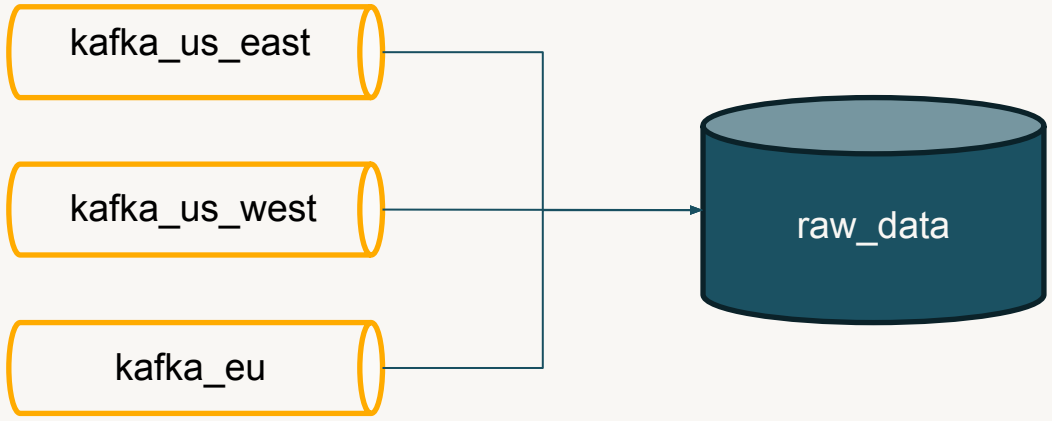
異なる名前のフローでストリーミング・ソースを進化させる

```
CREATE STREAMING TABLE raw_data


CREATE FLOW kafka_us_east
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)


CREATE FLOW kafka_us_west
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)


CREATE FLOW kafka_eu
AS INSERT INTO LIVE.raw_data BY NAME
SELECT * FROM kafka(…)
```



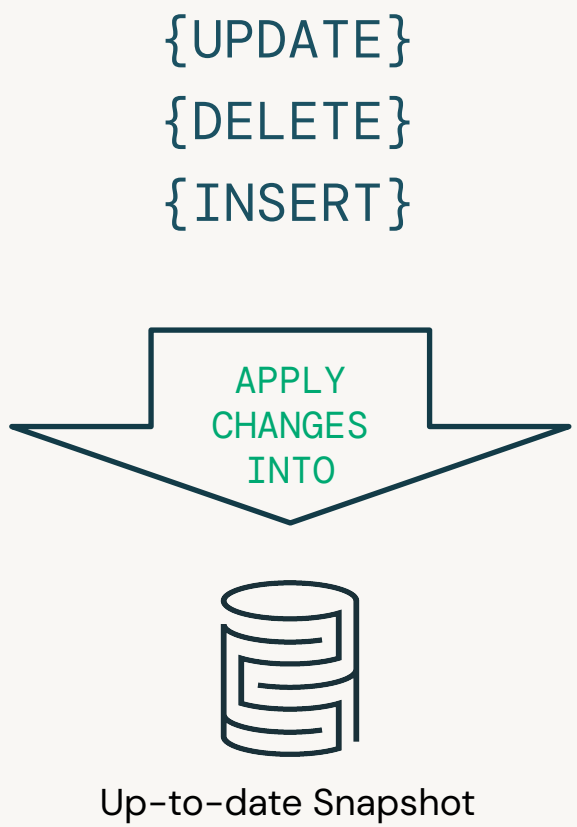
ストリーミング
チェンジデータキャプチャ(CDC)

CDC のための APPLY CHANGES INTO

他の場所に格納されているテーブルの最新のレプリカを保持する

```
CREATE STREAMING TABLE cities

APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```



CDC のための APPLY CHANGES INTO

他の場所に格納されているテーブルの最新のレプリカを保持する

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```

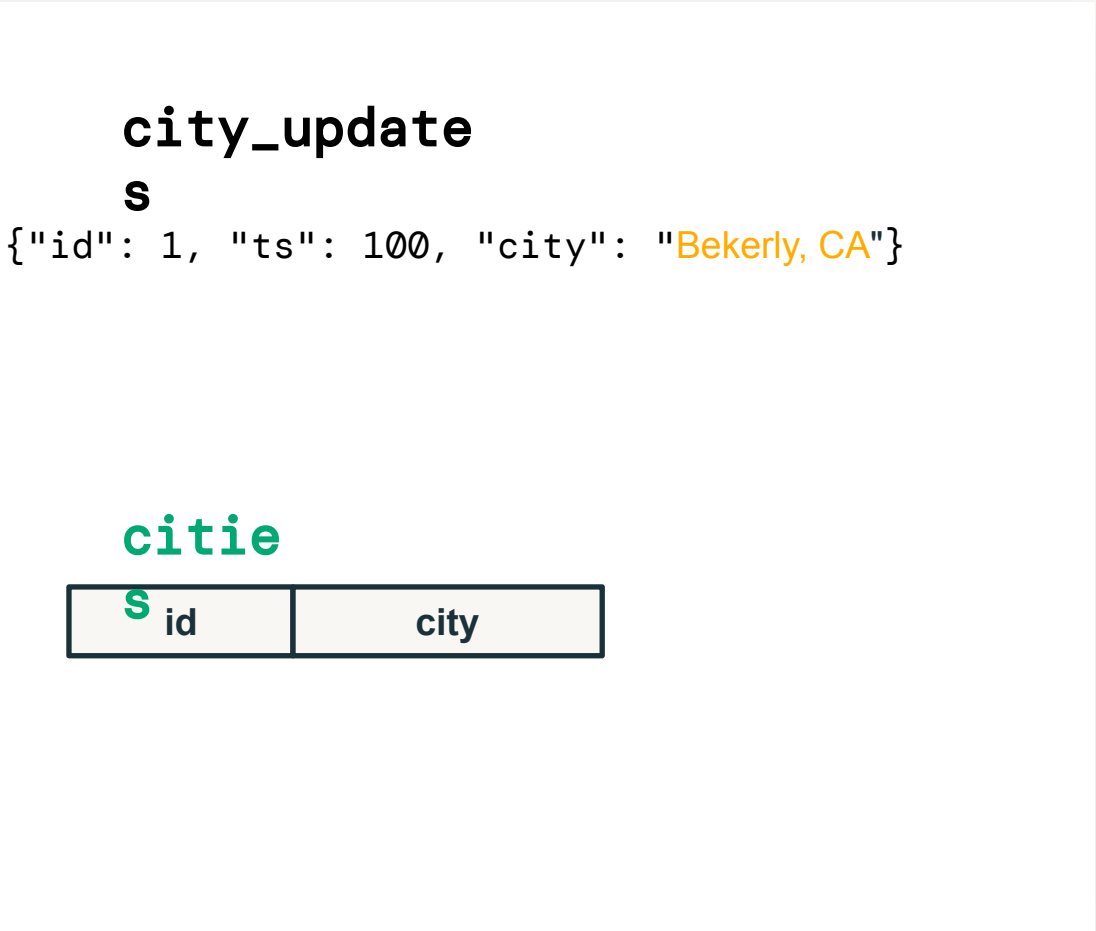
変更のソース、現在はストリームである必要がある

CDC のための APPLY CHANGES INTO

他の場所に格納されているテーブルの最新のレプリカを保持する

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```

変更を適用するターゲット

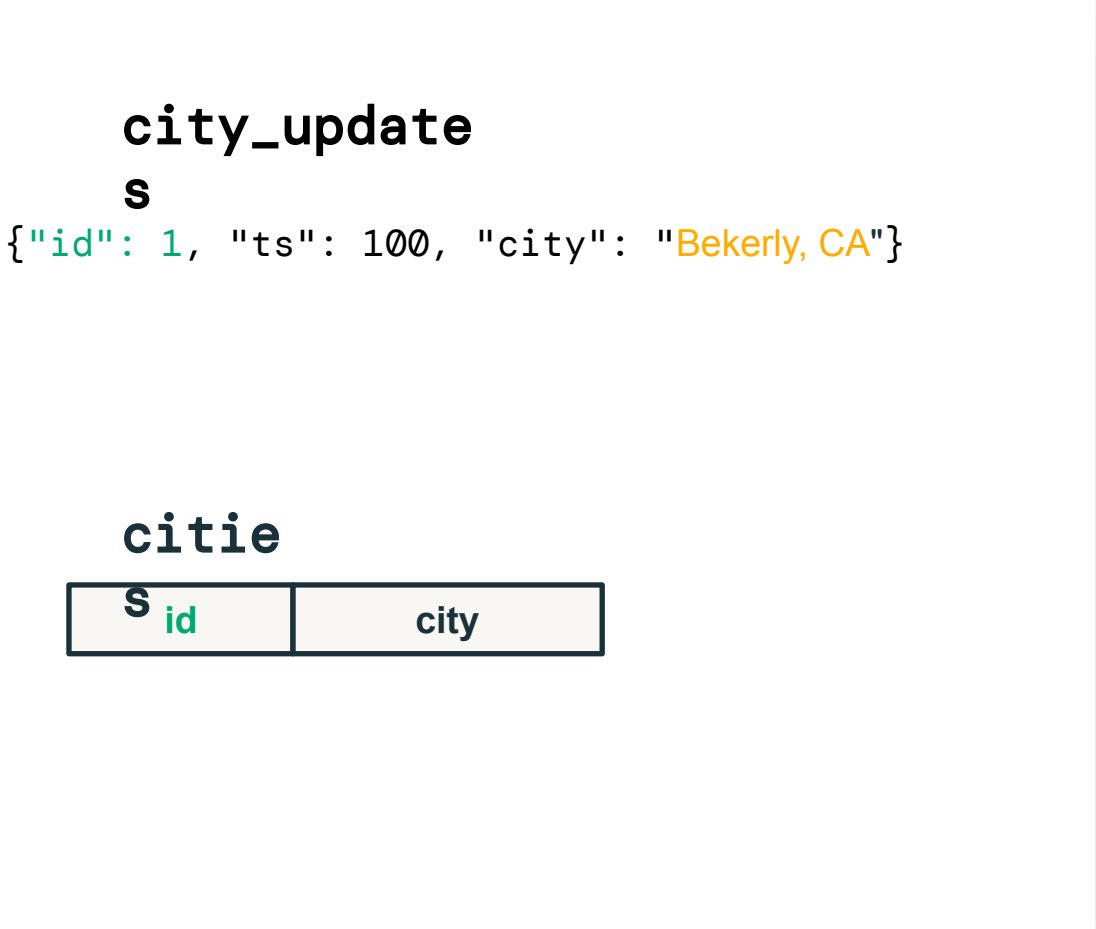


CDC のための APPLY CHANGES INTO

他の場所に格納されているテーブルの最新のレプリカを保持する

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```

指定した行を識別するための一意なキー



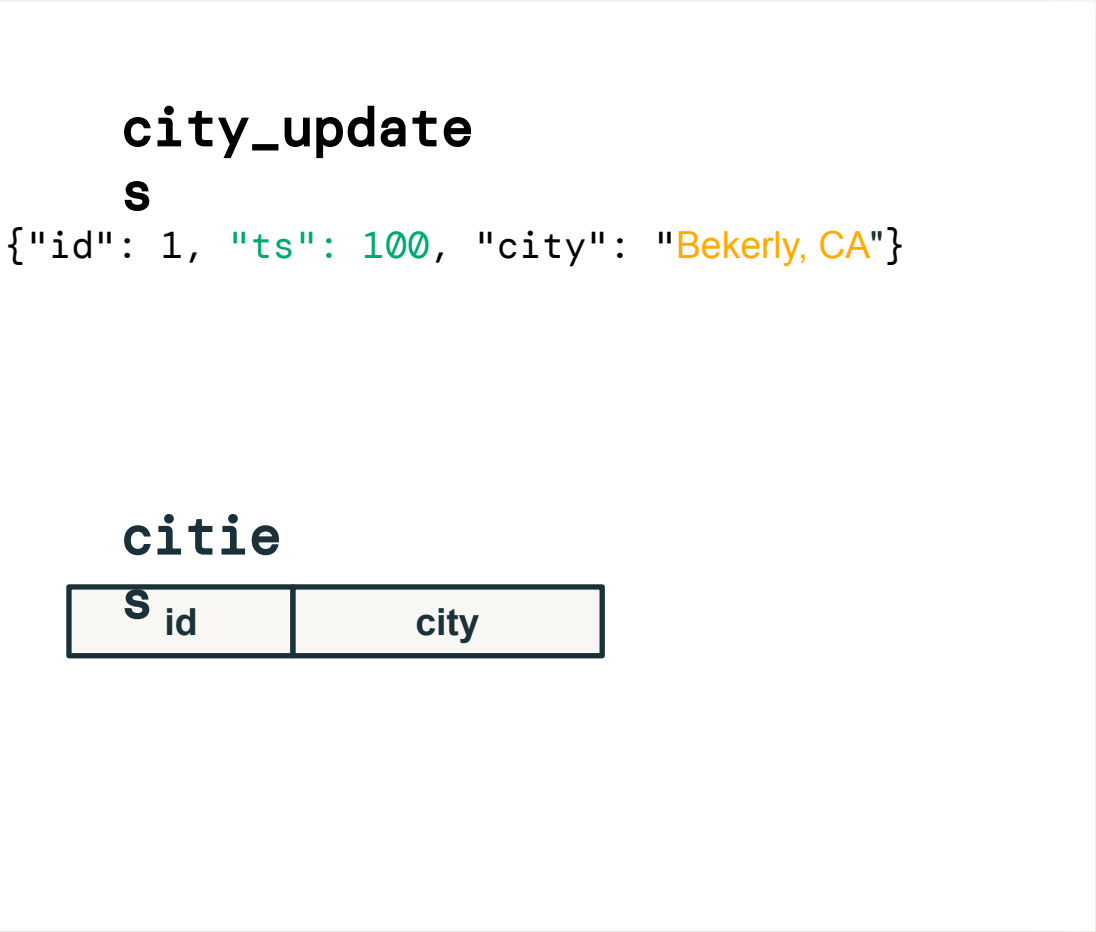
CDC のための APPLY CHANGES INTO

他の場所に格納されているテーブルの最新のレプリカを保持する

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(city_updates)
KEYS (id)
SEQUENCE BY ts
```

変更を順序付けるために使用できるシーケンス:

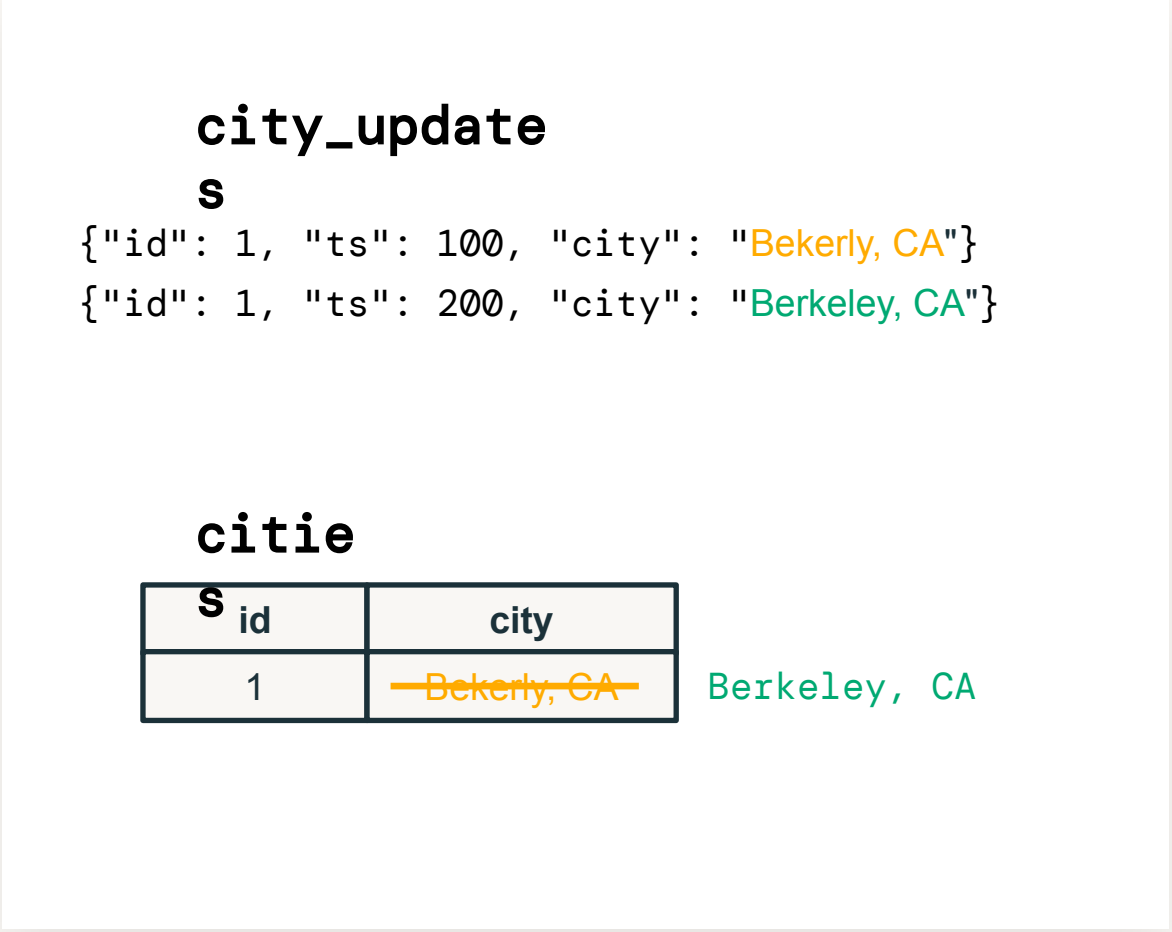
- ログシーケンス番号（lsn）
- タイムスタンプ
- 取り込み時間



CDC のための APPLY CHANGES INTO

他の場所に格納されているテーブルの最新のレプリカを保持する

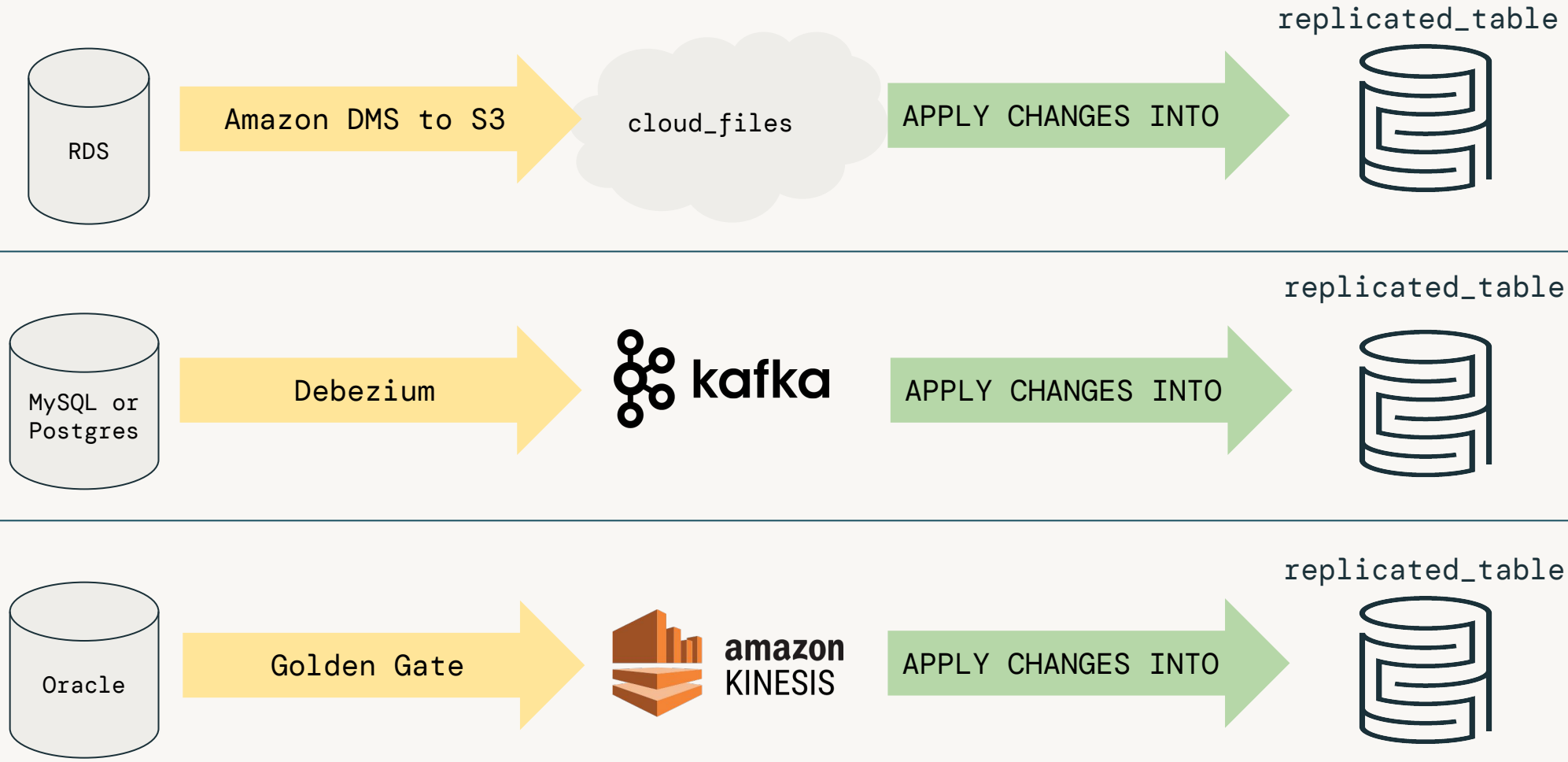
```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
```



REFERENCE ARCHITECTURE

RDBMSからのチェンジデータキャプチャ (CDC)

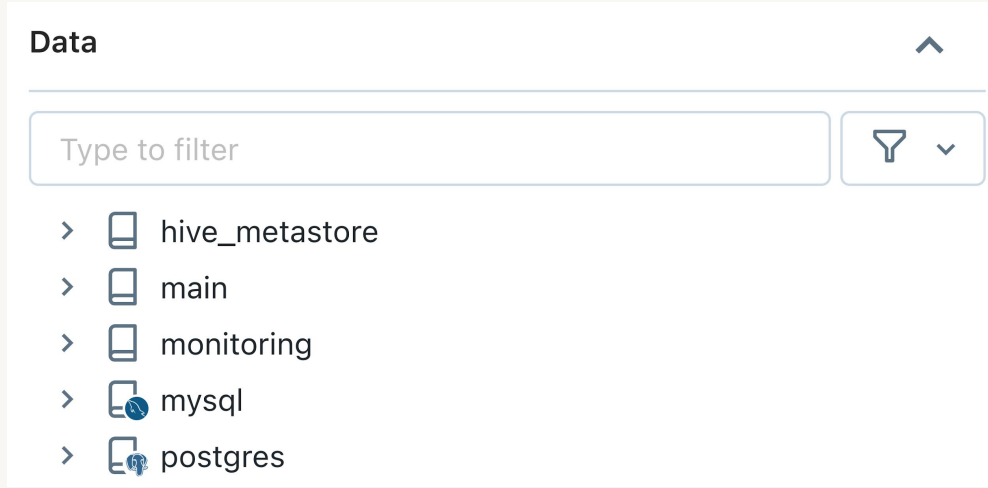
様々なサードパーティツールがストリーミング変更フィードを提供できる

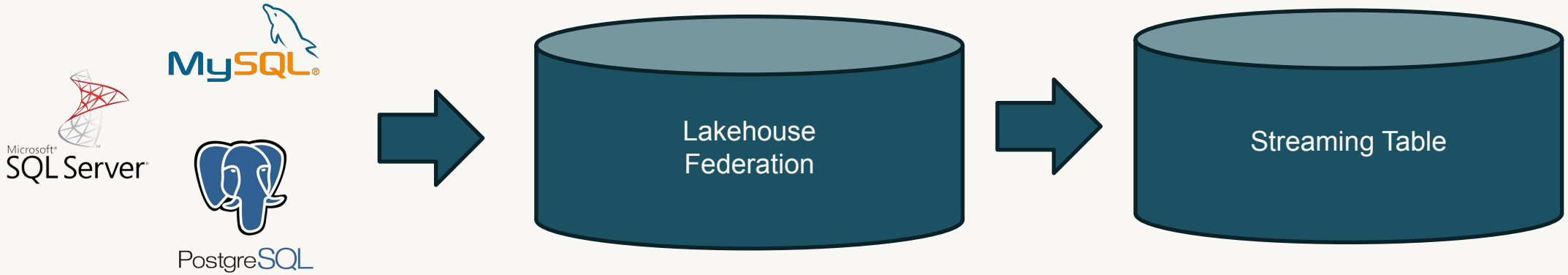


UCを使った Federated CDC

レイクハウスへのシームレスなレプリケーション

Unity Catalogを使用したCDCソースへのアクセス管理





CDC: どのように動くの？

調整（Reconciliation）により、異常データを透過的に処理が可能

Arrival Time

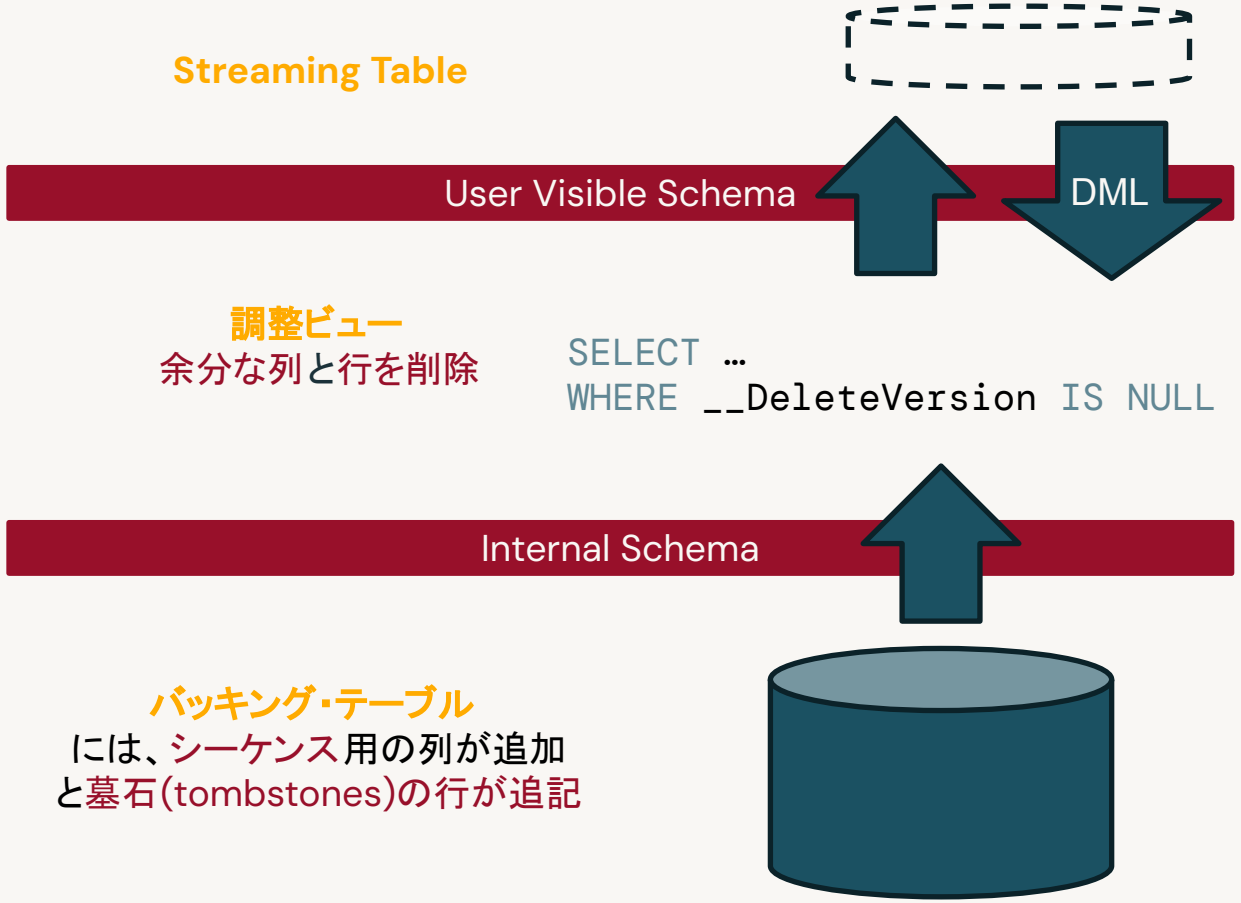
Id	timestamp	Operation
1	2	delete
1	1	update

Delete が実行される
ただしupdate後

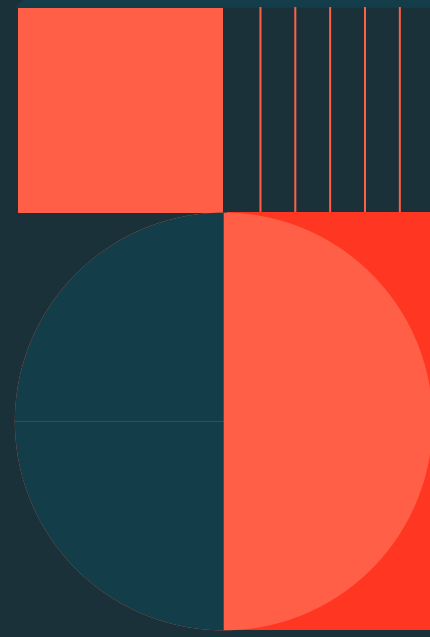
しかし update
の到着が遅れた

恒久的なゾンビ行を防ぐために、
deleteのタイムスタンプを 覚えてお
く必要がある。
ゾンビ行

図はDLT+UCを示す
HMSでは通常のビューで偽る必要がある



履歴を追跡するには？



緩やかに変化するディメンション Type 2

時間の経過とともに値がどのように変化したかの記録を保持

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS (id)
SEQUENCE BY ts
STORED AS SCD TYPE 2
```

__starts_at と __ends_at は SEQUENCE BY フィールド（ts）と同じ型になります。

city_updates

```
{"id": 1, "ts": 1, "city": "Bekerly, CA"}
{"id": 1, "ts": 2, "city": "Berkeley, CA"}
```

cities

id	city	__starts_at	__ends_at
1	Bekerly, CA	1	2
1	Berkeley, CA	2	null

ストリーミングを連結すべき時はいつか？

コストとレイテンシーのためのマルチホップストリーミング

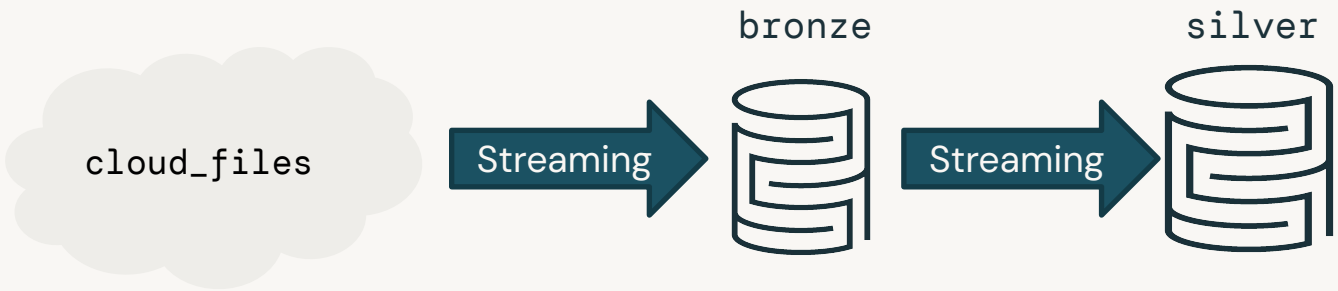
アペンドのみのデルタテーブルはソースにもシンクにもなれる

```
CREATE STREAMING TABLE bronze
AS SELECT * FROM cloud_files("/data/", "json”)


CREATE STREAMING LIVE TABLE silver
AS SELECT …
FROM STREAM(bronze)
```

複数ストリーミング・ジョブを連結させることで、以下のようなワークロードに対応：

- 非常に大きなデータ
- 非常に低遅延なターゲット



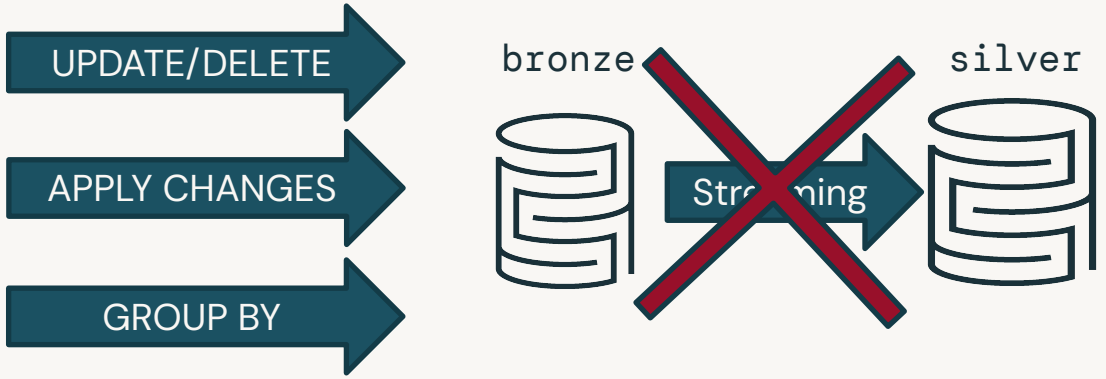
落し穴: ストリーミングパイプラインの更新

構造化ストリーミングは、アペンドのみの入力ソースを想定している

ストリーミング入力テーブルへの更新は、下流の計算を破壊する

- ストリーミングテーブルに対して実行される挿入以外のDML
- ウォーターマークなしで集計を計算するストリーミングテーブル
- APPLY CHANGES INTOと共に使用されるストリーミングテーブル

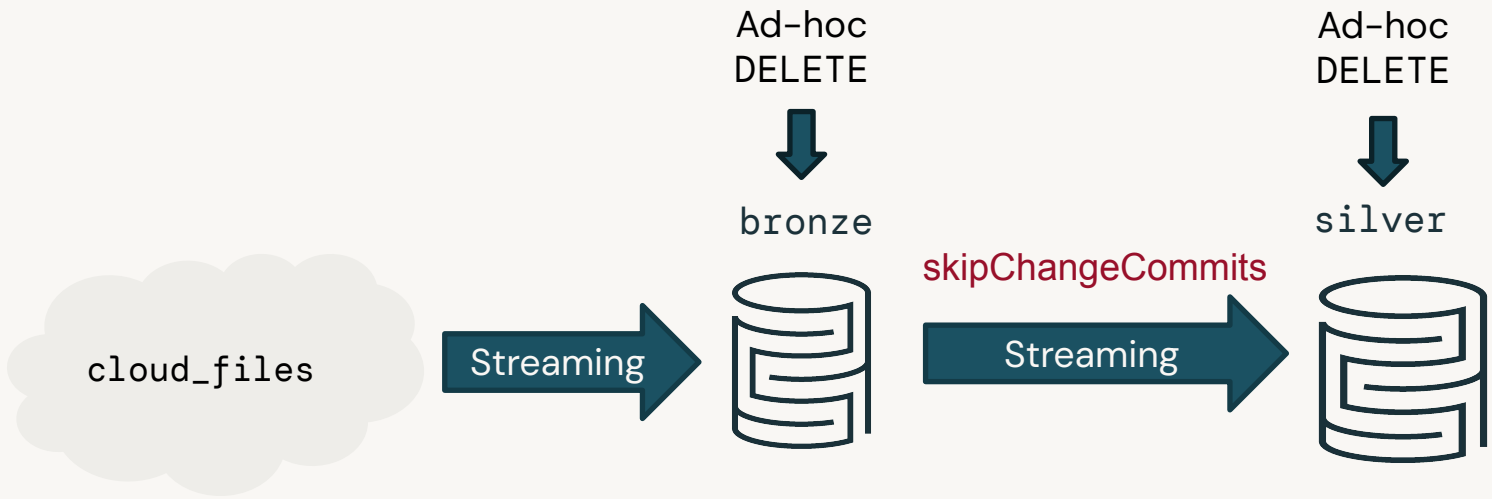
単発のDML実行後にストリームを修復するには、フルリフレッシュを使用



ストリーミングで更新を手動で行う

skipChangeCommits は変更後にパイプラインが失敗するのを防ぐ

- もし変更が珍しいものならskipChangeCommitsは失敗するのではなく、変更をスキップするようにストリーミングに指示する
- 手動で変更を下流に伝えるには DML を使用しなければならない。



Deep dive into Materialized Views

マテリアライズド・ビュー　変換の簡素化

マテリアライズド・ビューは、常に定義されたクエリと同じ結果を返す

ストリーミングテーブルと比較して、マテリアライズド・ビューは：

- 入力に対する appends, updates, deletes を処理します
- あらゆる集計/ウィンドウ関数をサポート
- ad-hoc クエリと同じように結合もサポート
- 自動的にクエリ定義の変更を処理

ストリーミングとは異なり、マテリアライズド・ビューは 入力を複数回読み取ることが許可され、必要に応じて最初から再計算することが可能

Enzymeを使用したマテリアライズド・ビューの増分リフレッシュ

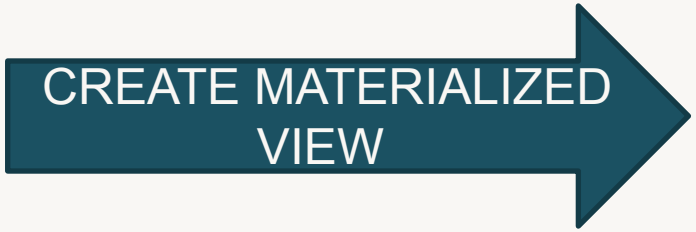
データはいつも変化している

インクリメンタル化は、すべてを再計算することなく、派生データセットを更新

date	city_id
2022-06-01	1
2022-06-01	2
2022-06-02	3

date	city_id	city_name
2022-06-01	1	Berkeley
2022-06-01	2	San Francisco
2022-06-01	3	Denver

date	city_id
2022-06-01	1
2022-06-01	2
2022-06-02	3



date	city_id	city_name
2022-06-01	1	Berkeley
2022-06-01	2	San Francisco
2022-06-01	3	Denver

どのようにEnzymeは
異なるタイプのクエリーを
増分リフレッシュできるのか？

PREVIEW

新しいデータが到着するたび追加する

新しいデータが追加され、クエリが単調である場合に機能します

mon·o·ton·ic que·ry

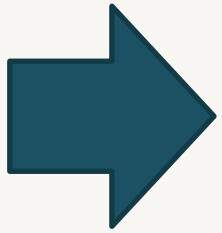
/ˌmänəˈtänik ˈkwirē/

noun

データベースに新しいタプルが追加され、以前に出力したタプルを一切失わない問い合わせ

- 非常に効率的 (ストリーミングのような!)
- select/project/inner join/その他に対して動作
- 入力に対する変更を行わない

date	city_id
2022-06-01	1
2022-06-01	2
2022-06-02	3

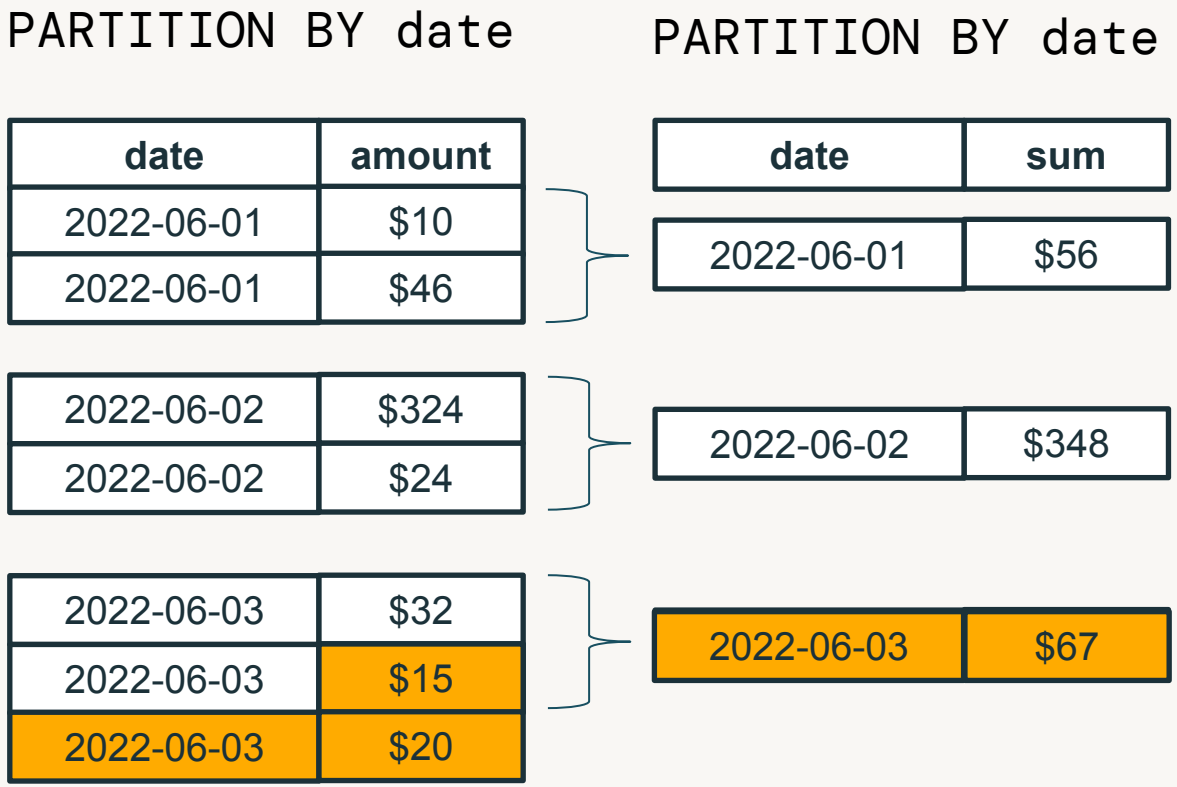


date	city_id	city_name
2022-06-01	1	Berkeley
2022-06-01	2	San Francisco
2022-06-02	3	Denver

PREVIEW

Partition 再計算

テーブルをパーティションに分割し、変更されたものだけを再計算する



- 集約と更新を扱うことが可能
- 入力と出力が同じパーティショニングであることが必要

PREVIEW

MERGE 特定の行を更新

データベース文献のテクニックを使って、結果の変化を計算する

- 複雑なクエリーを処理
- update/insert/deleteを扱える
- マージは高価
- 推論が複雑

date	amount
2022-06-01	$10
2022-06-01	$46
2022-06-02	$324
2022-06-02	$24
2022-06-03	$32
2022-06-03	$15
2022-06-03	$20

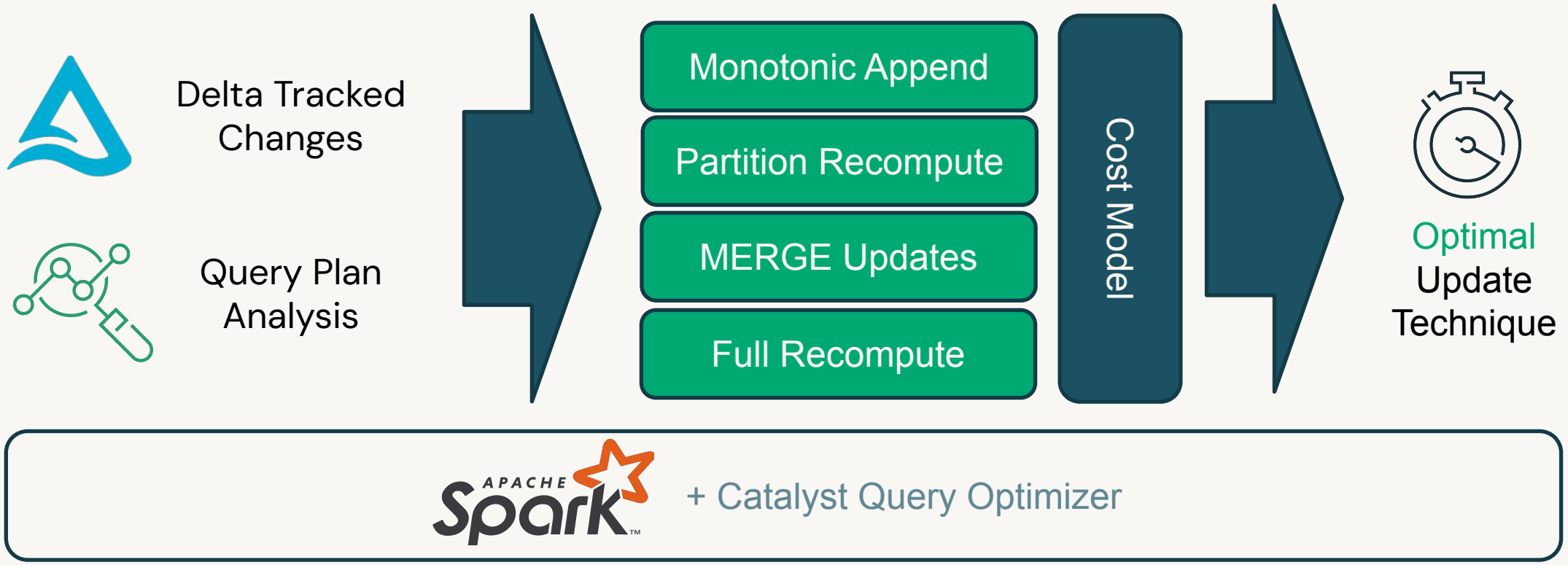
date	sum
2022-06-01	$56
2022-06-02	$348
2022-06-03	$67

Enzyme は
正しいテクニックを
どう選ぶのか？

PREVIEW

Enzyme の紹介

自動でのインクリメンタルETL最適化

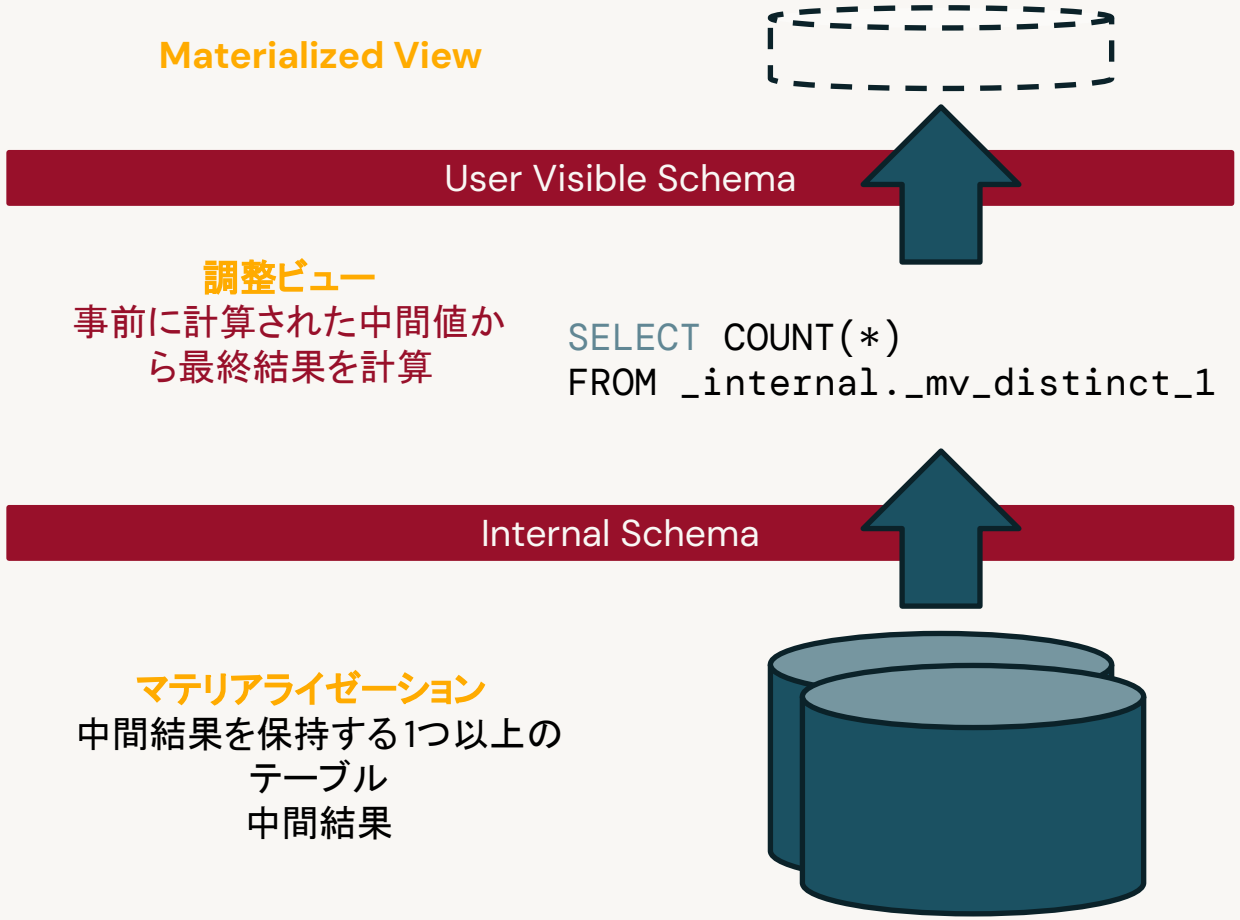


Enzyme: どう動く？

分解(Decomposition)はクエリを分割し、インクリメンタル化を可能に

```
CREATE MATERIALIZED VIEW total_unique_users
SELECT COUNT(DISTINCT customer_id)
FROM prod.clicks
```

```
CREATE MATERIALIZED VIEW _internal._mv_distinct_1
SELECT COUNT(*)
FROM prod.clicks
GROUP BY customer_id
```



アナウンス: DLT Serverless

PREVIEW

DLT ServerlessはEnzymeがデフォルトで有効

Enzymeは以下でのみ使用可能:

- DBSQL で作成された MV (Public Preview)
- DLT Serverless パイプラインにおけるMV（Private Preview）

Enzymeは多くのクエリをインクリメンタルにリフレッシュ可能:

- Co-partitioned
- Associative aggregates
- Monotonic queries

Enzyme ロードマップ

- Inner/outer joins
- Semi-ring aggregates
- Distinct aggregates
- Window functions
- Predicates on current_date()

Demo

https://www.databricks.com/resources/demos/tutorials/lakehouse-platform/full-delta-live-table-pipeline

Thank you