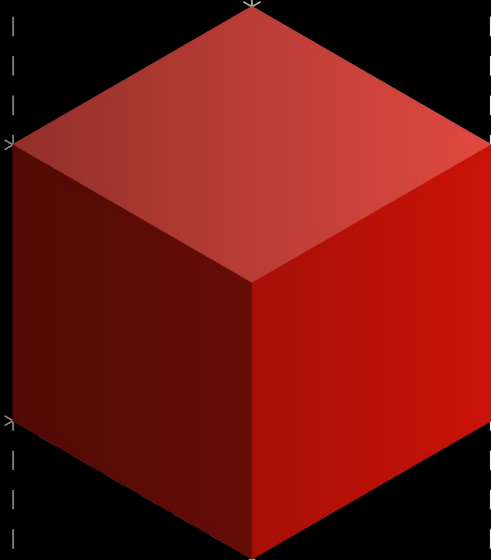




分散SQLデータベース「TiDB」の リアルな課題と運用ノウハウ

Koitabashi Yoshitaka
Senior Solution Architect / PingCAP Japan

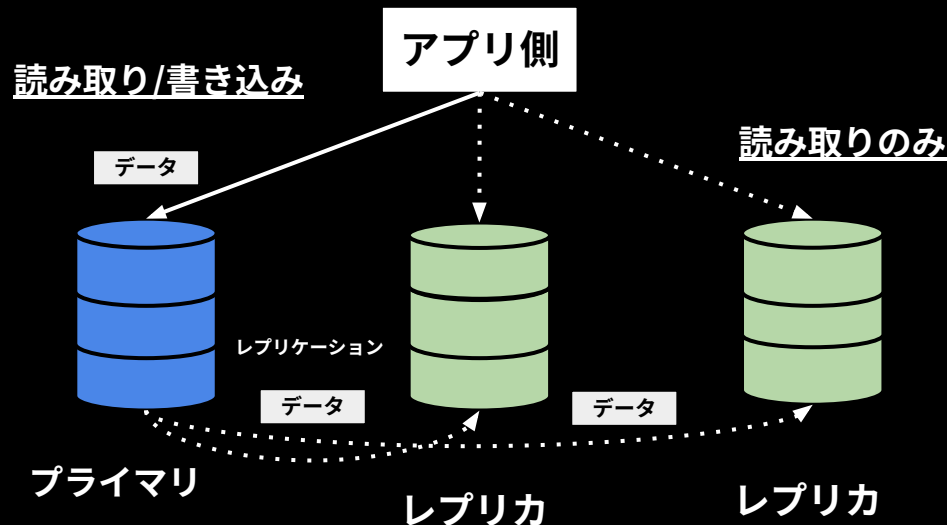
11 Jul. 2025



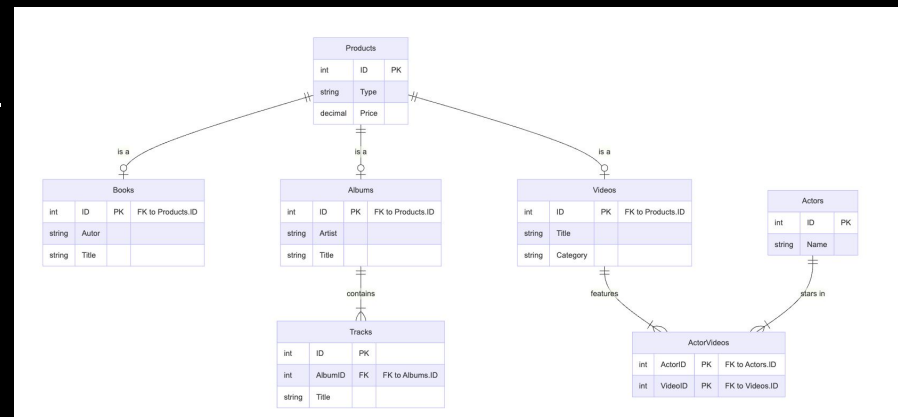
- 01 分散型SQLデータベース
 - 02 TiDBの裏側
 - 03 実際の運用現場で直面する課題
-

01 分散型SQLデータベース ≡ NewSQL

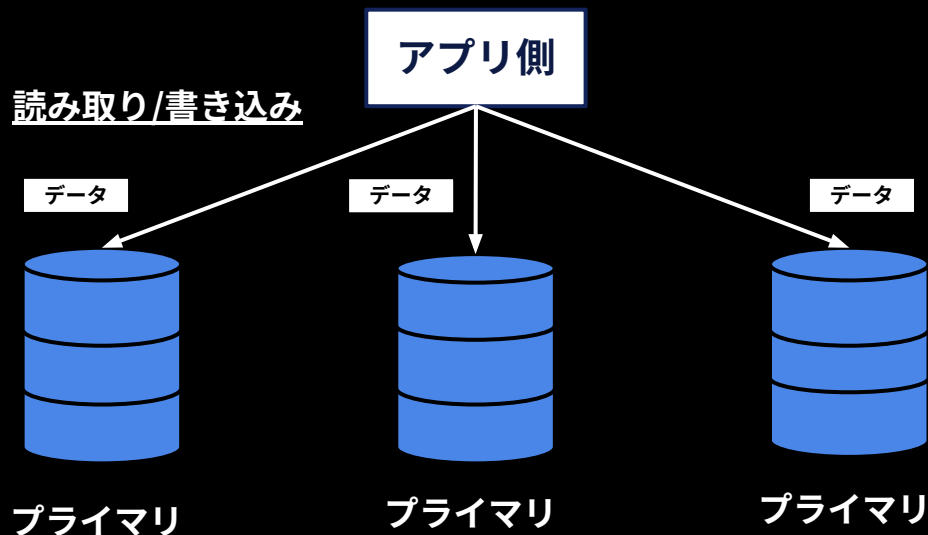
例: RDBMSアーキテクチャ / テーブル構造



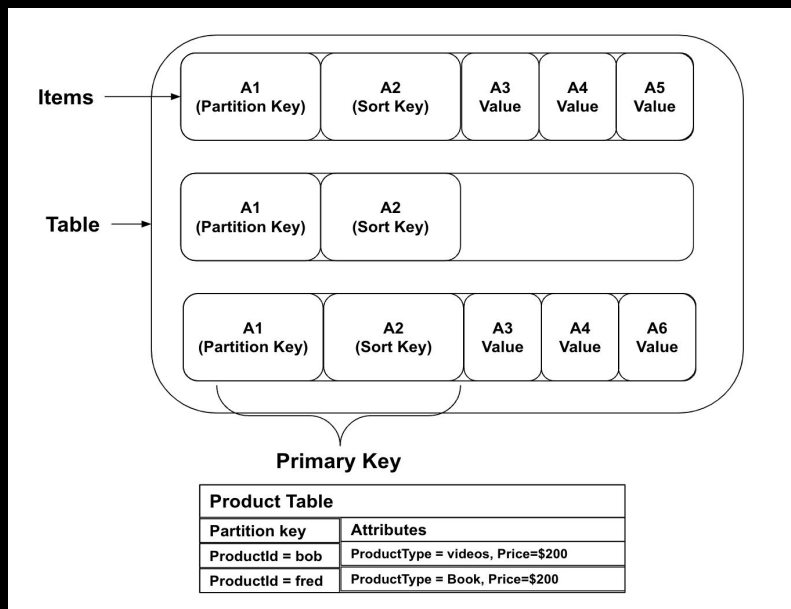
例: 製品(Products)データベース



例: NoSQLアーキテクチャ / テーブル構造



例: KVSベースのテーブル設計



例: 分散型SQLデータベース ≡ NewSQL

アプリ側

読み取り/書き込み

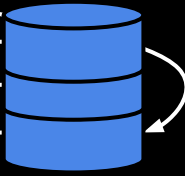
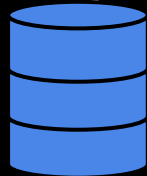
コンピューティングノード

ノードA

ノードB

ノードC

ストレージノード

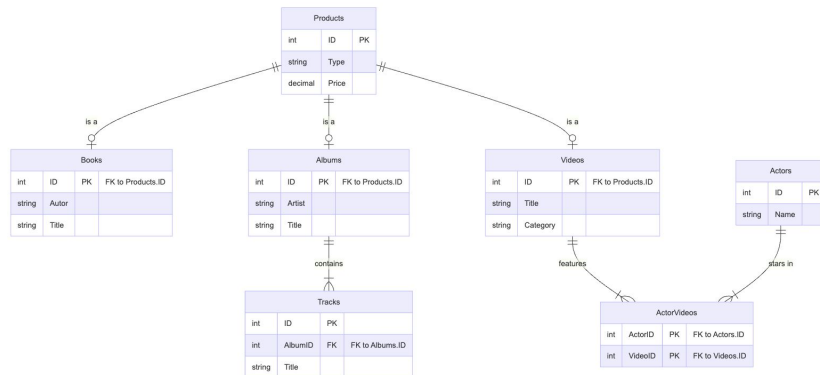


プライマリ

プライマリ

プライマリ

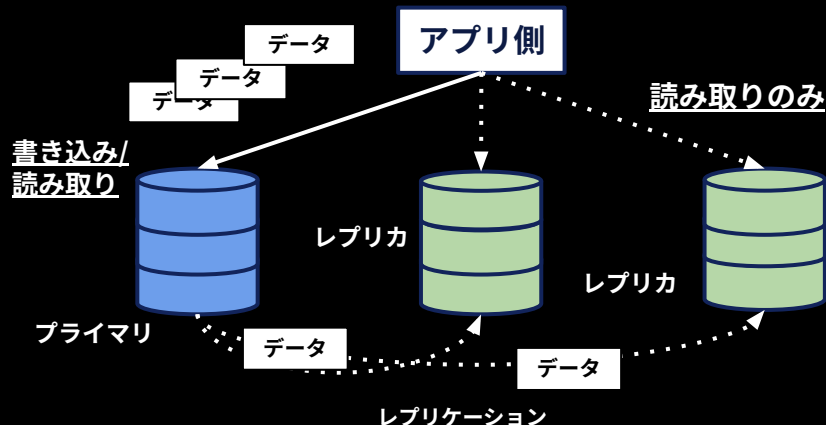
例: 製品(Products)データベース



RDBMSとNewSQL(TiDB)との違い

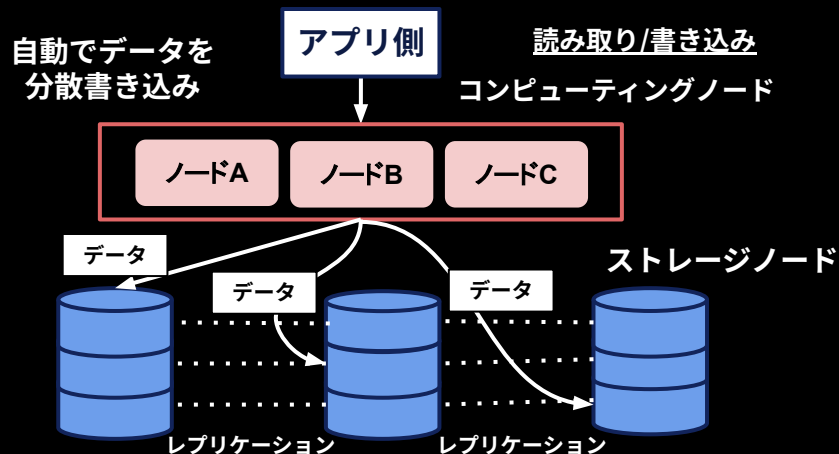
RDBMS

- 書き込み負荷の集中に伴うボトルネック
- レプリカのスケール限界 / レプリカラグ

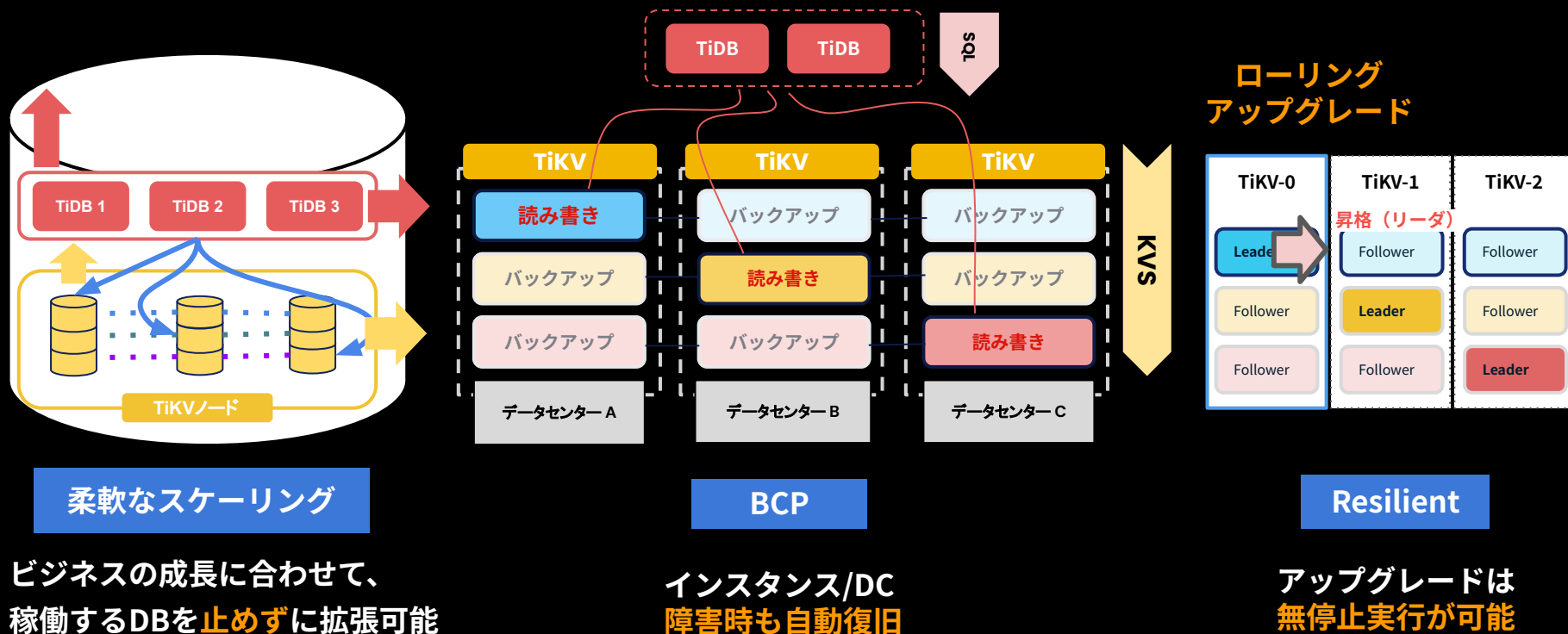


NewSQL (TiDB)

- 大量の同時接続でも一定のレイテンシ
- オンラインで水平/垂直に拡張可能

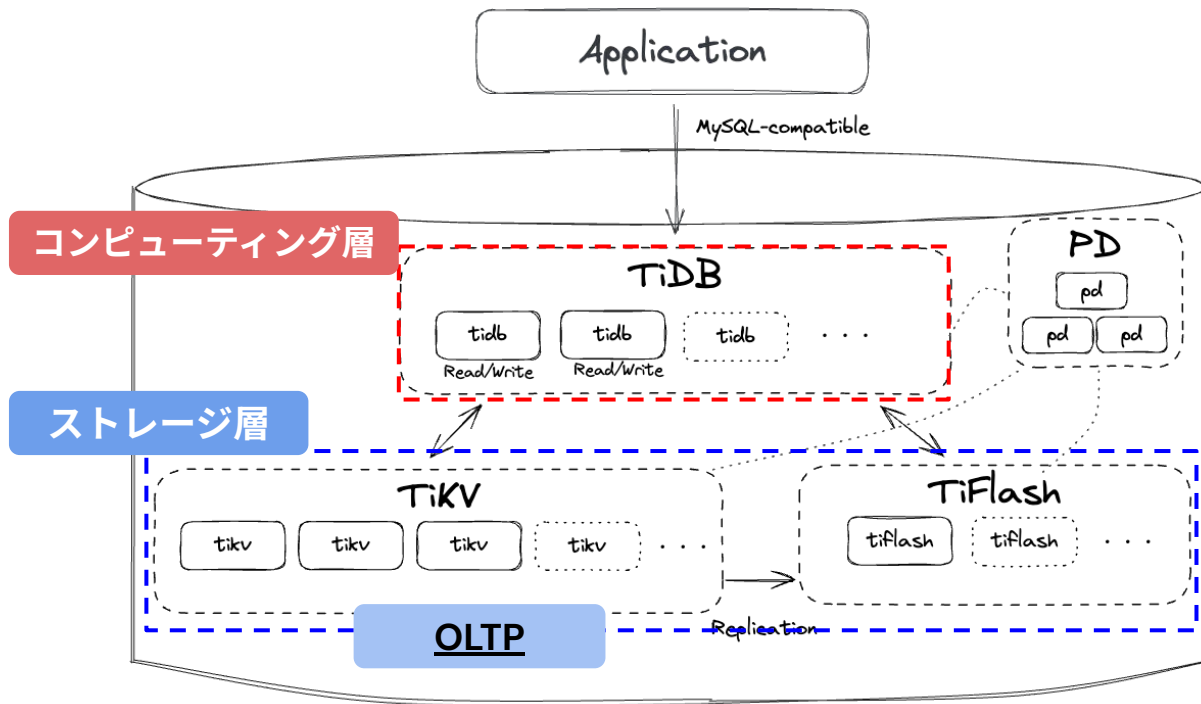


なぜ分散型SQLデータベースが注目されているのか

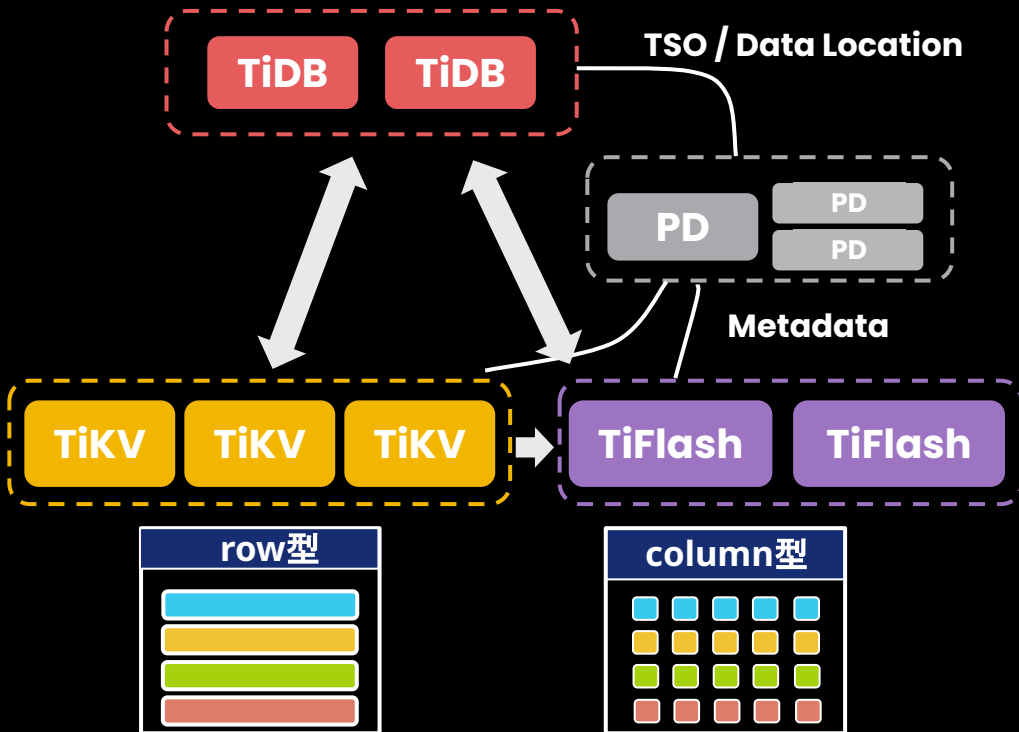


02 TiDBの裏側

全体のアーキテクチャー



TiDB内部のコンポーネント



コンピューティング

TiDB

- SQL解析を行う *MySQLプロトコルをKVに変換
- HTAPではTiKV(row型)かTiFlash(column型)判断

ストレージ

TiKV

TiFlash

TiKV

- OLTP用途の分散Key-Value Store※RocksDB利用
- パーティショニングによるデータ分散管理
- Raftや2PCにより整合性担保

TiFlash

*optional

- HTAP利用時の追加コンポーネント
- OLAP用途のカラム型ストア

司令塔

PD

- データ配置場所の管理 (Region管理)
- 分散トランザクションで利用するTSOを発行

よくあるOLTPワークロード / OLAPワークロード

商品を買いたいユーザ

在庫保管している
倉庫作業員

商品を配送するトラック
のドライバー

小売系ECサイト

ECサービス

在庫状況確認
サービス

配送情報確認
サービス

OLTPシステム

ユーザ情報DB

商品在庫DB

配達場所管理DB

ETL

ETL

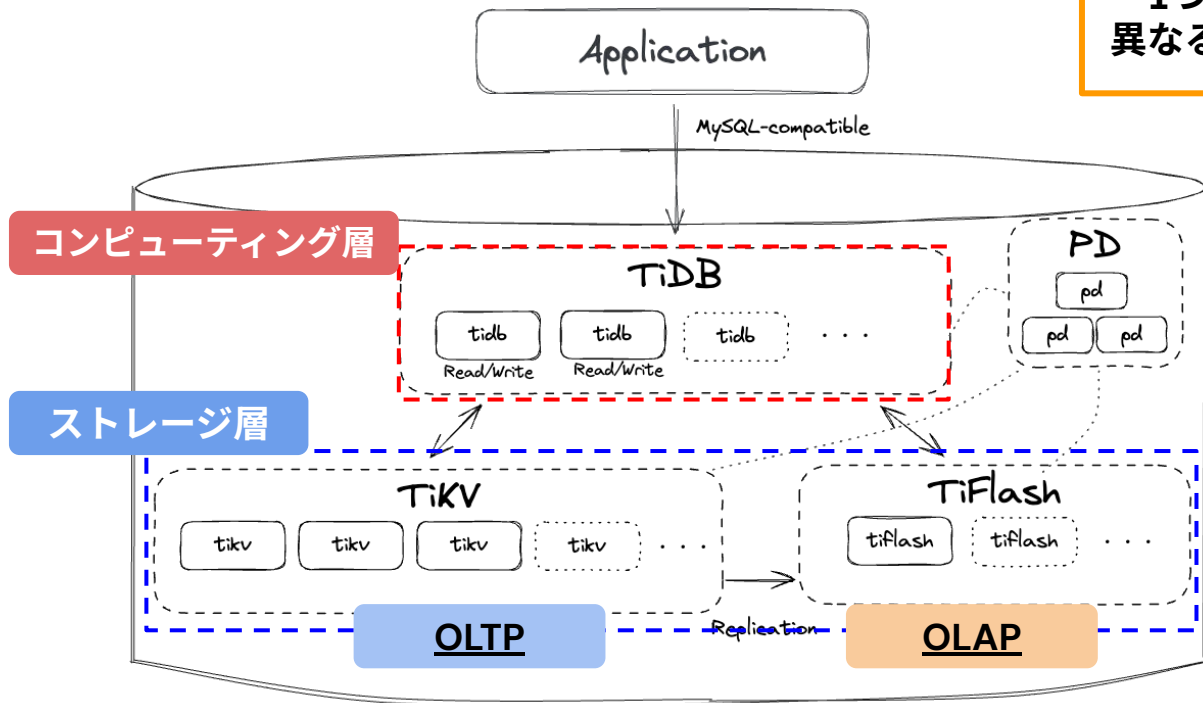
ETL

OLAPシステム

データウェアハウス

TiDBならではの機能としてHTAP

1つのDBクラスターで
異なるワークロードを実現

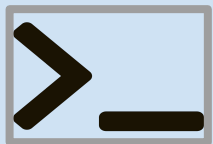


TiDBの利用パターン

パターン

1

Instance



TiUP

TiDBのセットアップから運用まで
※ローカルでも使える

パターン

2

Kubernetes



TiDB Operator

K8sでのセットアップ・運用

パターン

3

Cloud

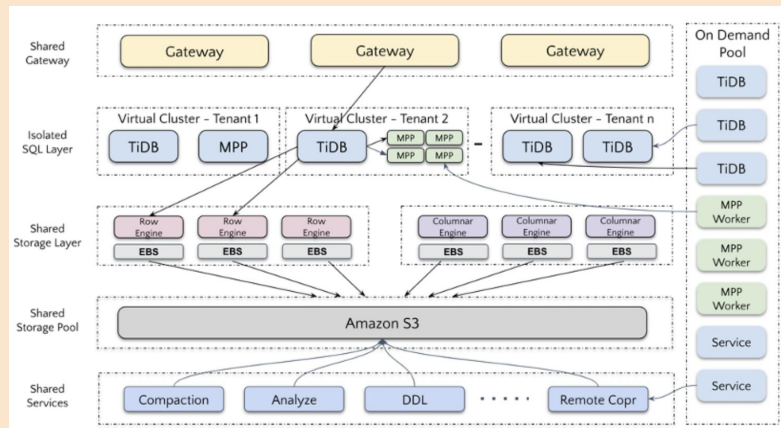


TiDB Cloud

フルマネージドDBaaS
ServerlessとDedicatedがある

TiDB Cloudのラインナップ

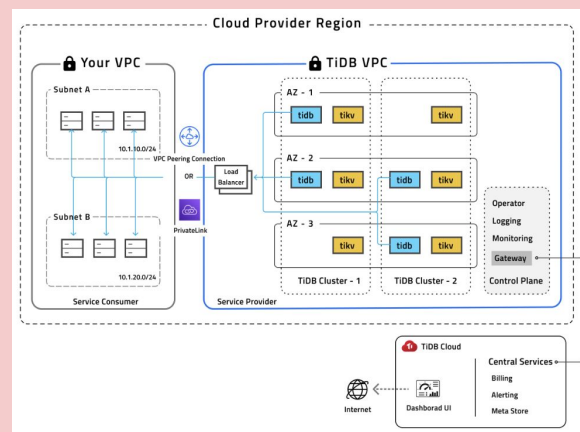
Serverless



マルチテナント型

- シングルAZ or マルチAZ / Autoスケール
- BranchingやEdge Function用のDriver用意
- Vector Search※Betaもあり

Dedicated



専有型

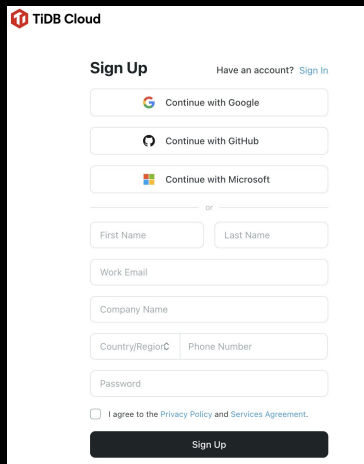
- マルチAZ / 各コンポーネントスケール(APIあり)
- MySQLからの移行ツールもフルで使える
- OSSライクに柔軟なチューニング可能

簡単に始められるTiDB Serverless

クレカ不要！
\$0から利用可能

- ・1ヶ月間で行ベースデータ5GiB / 列ベースデータ5GiBを同時に保存可能
- ・5,000万RUを消費可能
- ※無料クラスター: 1アカウント5つまで

MySQLクライアントからの
接続時の情報も自動生成



The image shows the TiDB Cloud Sign Up page. It features the TiDB Cloud logo at the top left. Below it, the 'Sign Up' heading is followed by a link 'Have an account? Sign In'. There are three social login buttons: 'Continue with Google', 'Continue with GitHub', and 'Continue with Microsoft'. Below these is an 'or' separator. The form includes input fields for 'First Name', 'Last Name', 'Work Email', 'Company Name', 'Country/Region', 'Phone Number', and 'Password'. At the bottom, there is a checkbox for 'I agree to the Privacy Policy and Services Agreement.' and a 'Sign Up' button.

クラスターの起動10秒くらい

pingcap-webinar-20250424

Serverless

● Available

Connect With

MySQL CLI

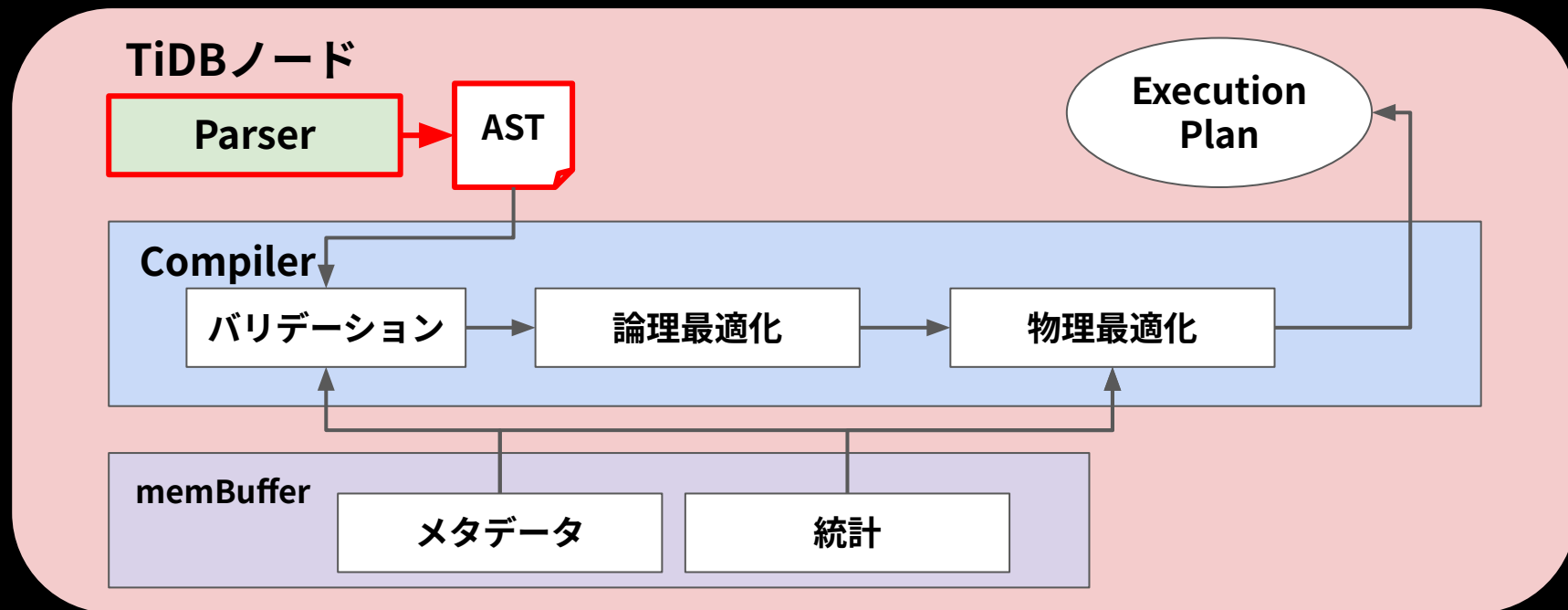
Existing connections are based on password you've set before. Forget it? [Reset Password](#)

```
mysql --comments -u '[redacted].root' -h gateway01.ap-northeast-1.prod.a  
ws.tidbcloud.com -P 4000 -D 'test' --ssl-mode=VERIFY_IDENTITY --ssl-ca=/etc/s  
sl/cert.pem -p'<PASSWORD>'
```

Connections to TiDB Serverless clusters with public endpoint require TLS. Learn more about [secure connection settings](#).

分散DBの裏側

TiDBの役割: SQLの変換とトランザクション



TiDBにおけるテーブルからKey-Valueへの変換

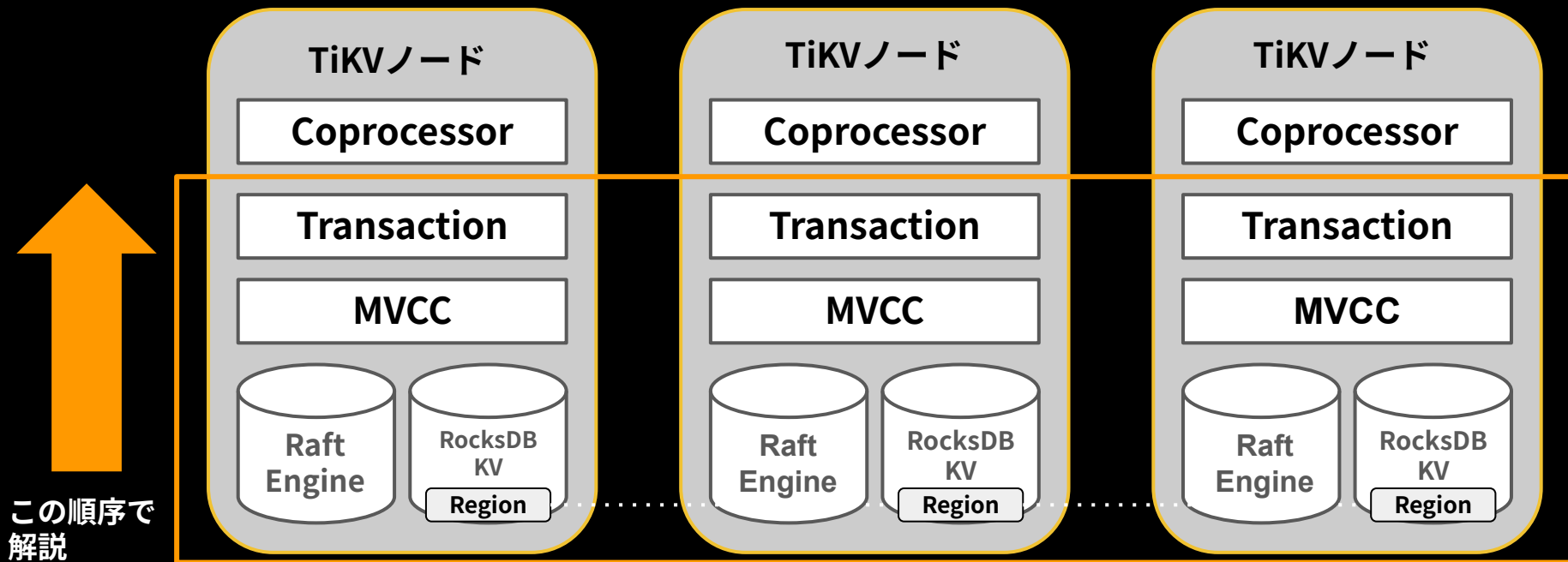
- 行データを <{table_id, primary key},{values*}> の形のKVに変換
- インデックスも <{index_columns*},rowid>の形のKVに変換

id	名前	誕生日	携帯電話	点数
r1	Tom	1982-09-28	1390811212	78
r2	Jack	1996-04-12	1801222187	91
r3	Frank	1982-09-28	1364571212	90
r4	Tony	1977-03-12	1391113134	65
r5	Jim	1992-07-19	1579915611	51
r6	Sam	1978-09-12	1713665011	97

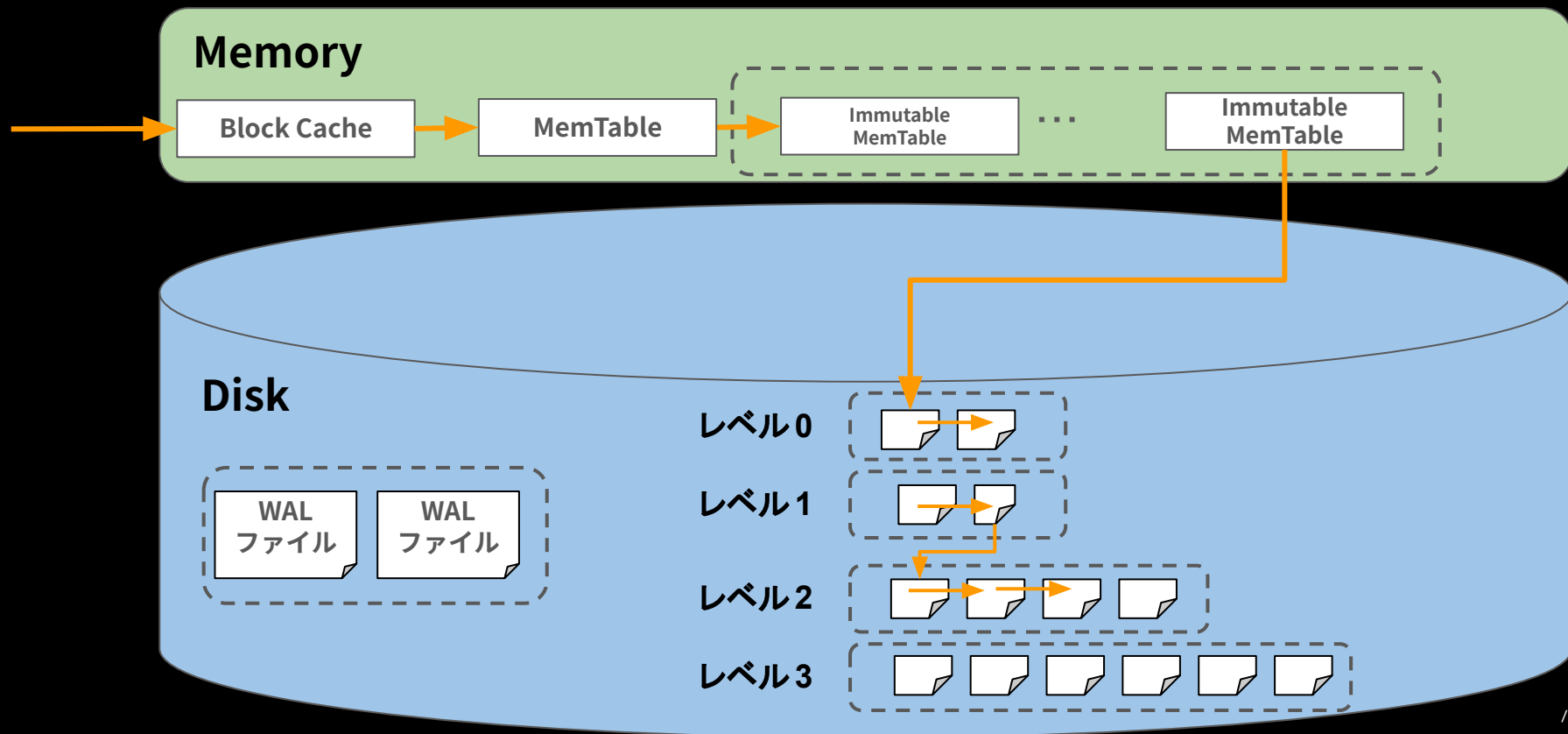


Key	Value
tid_r1	Tom, 1982-09-28, 1390811212, 78
tid_r2	Jack, 1996-04-12, 1801222187, 91
tid_r3	Frank, 1982-09-28, 1364571212, 90
tid_r4	Tony, 1977-03-12, 1391113134, 65
tid_r5	Jim, 1992-07-19, 1579915611, 51
tid_r6	Sam, 1978-09-12, 1713665011, 97

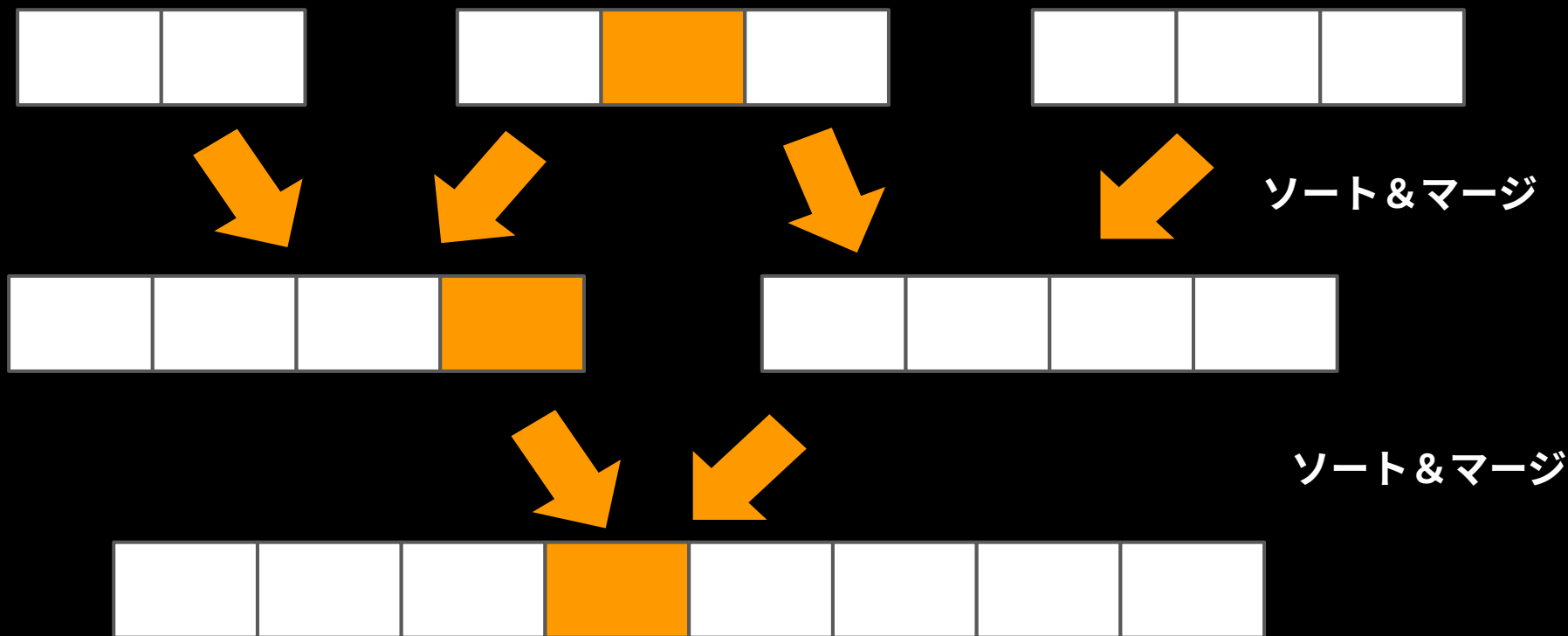
TiKVの役割: データの保存と一貫性



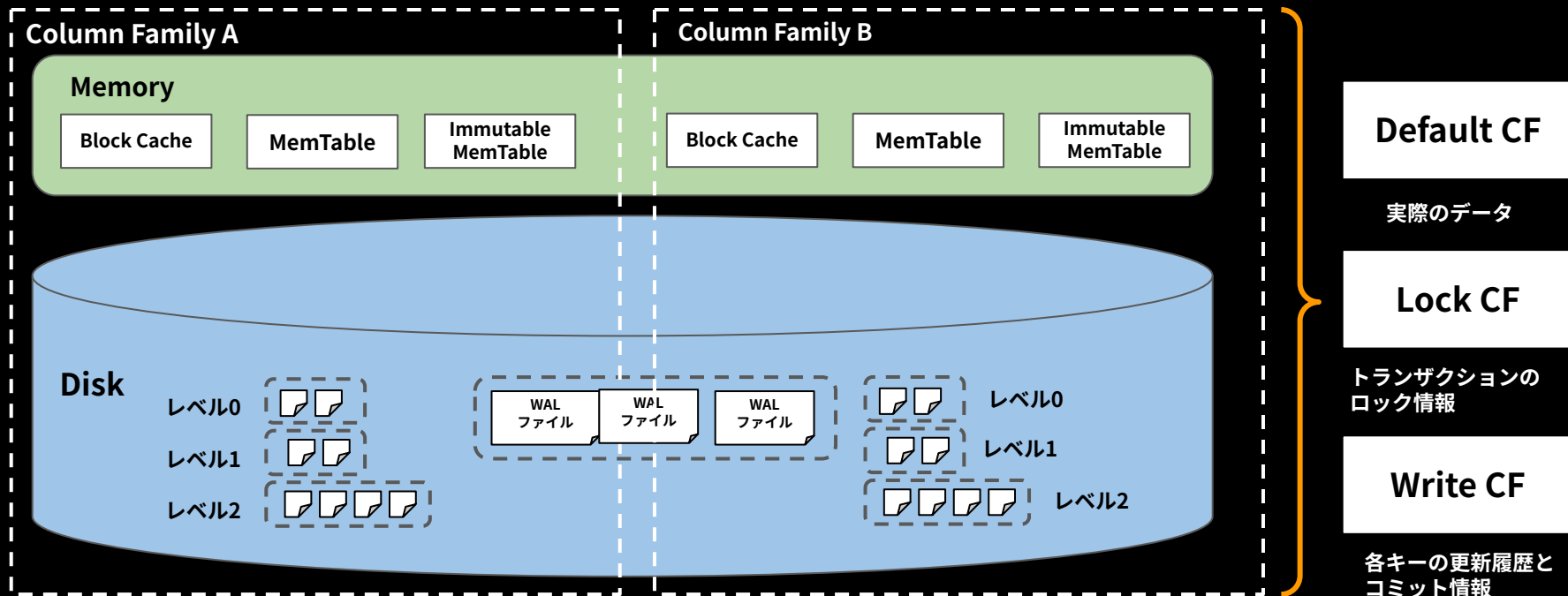
RocksDBの書き込み処理の特徴



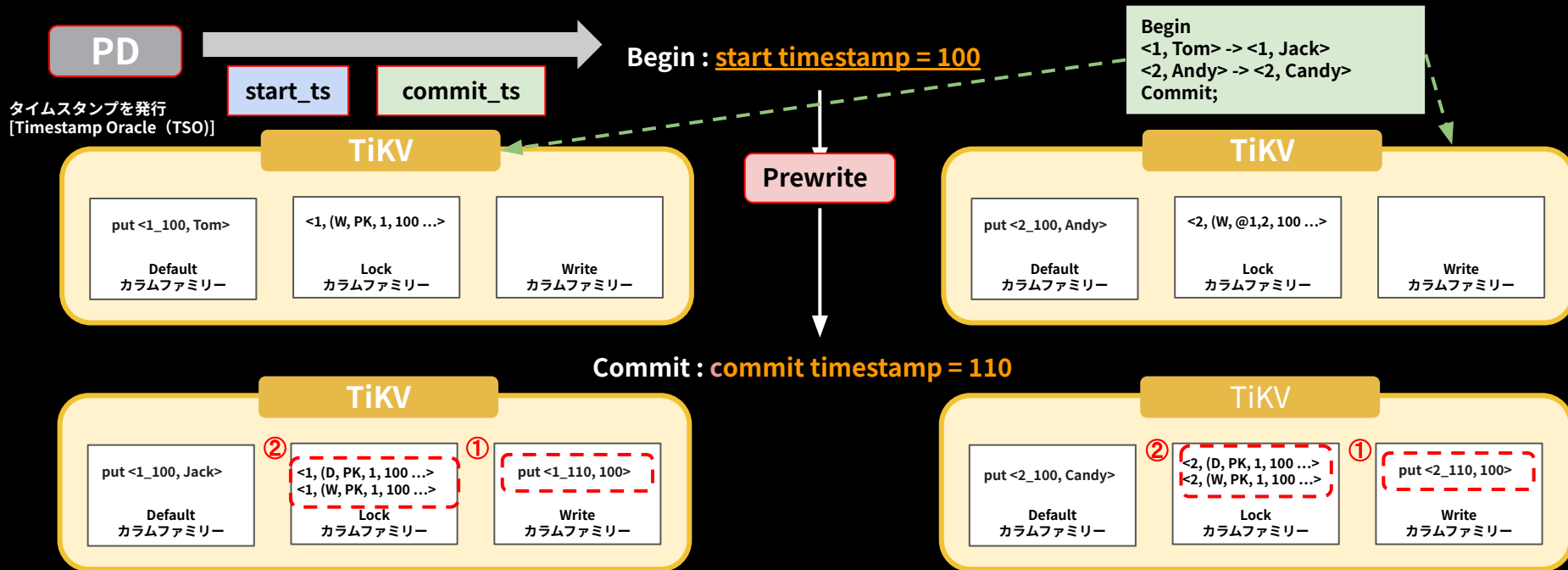
ログ構造化マージ(LSM)ツリーの動き



RocksDBにおけるCF(カラムファミリー)



分散トランザクション / MVCC



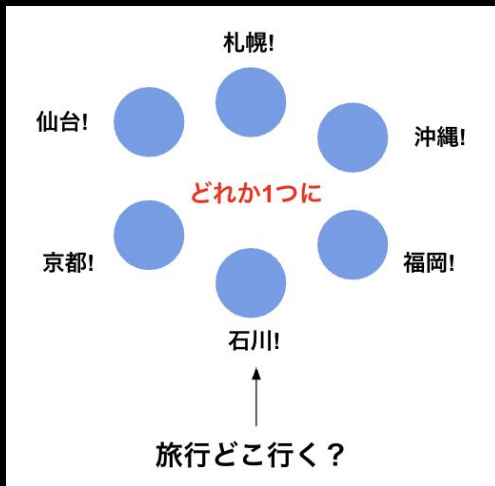
レプリカによる**可用性**: Raft

分散システム: **Consensus(合意)**形成は重要

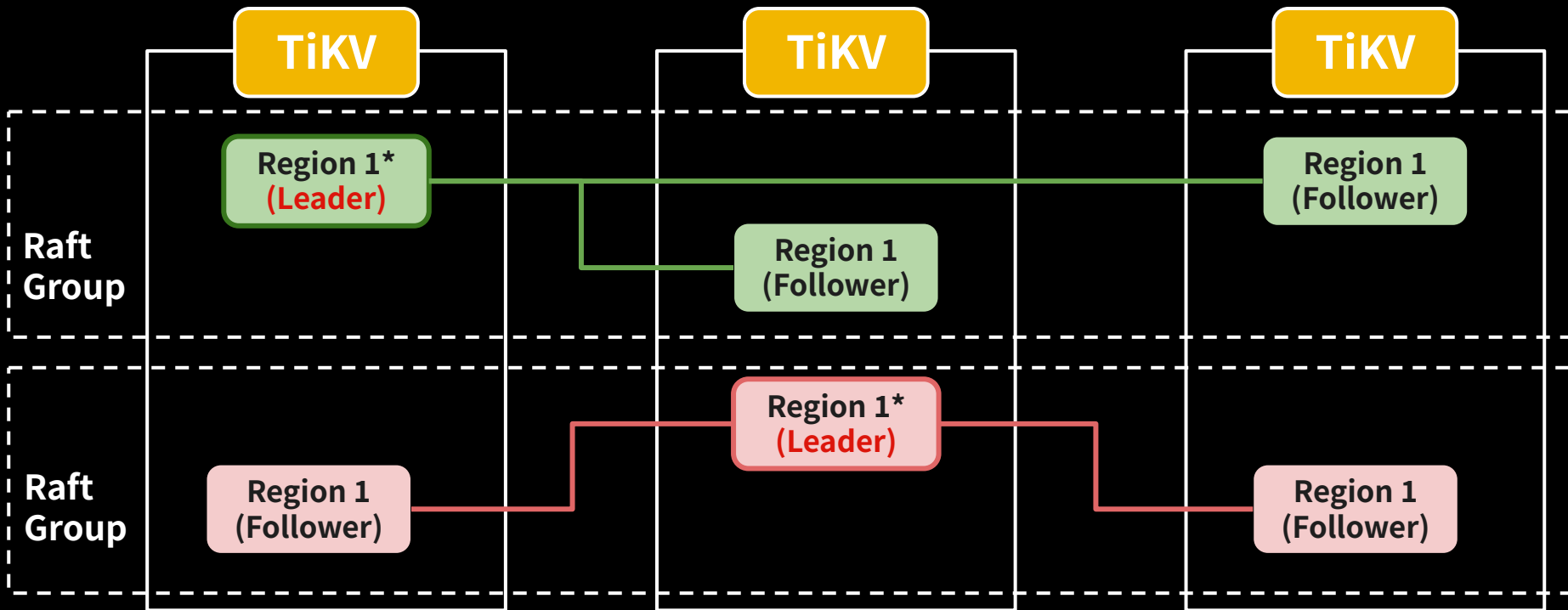
- 異なるレプリカを作り出さないようにする
- リーダーの決定や分散トランザクションに利用

一貫性を保つ合意アルゴリズム

- Raft => TiDBではこちらを採用
- Paxos

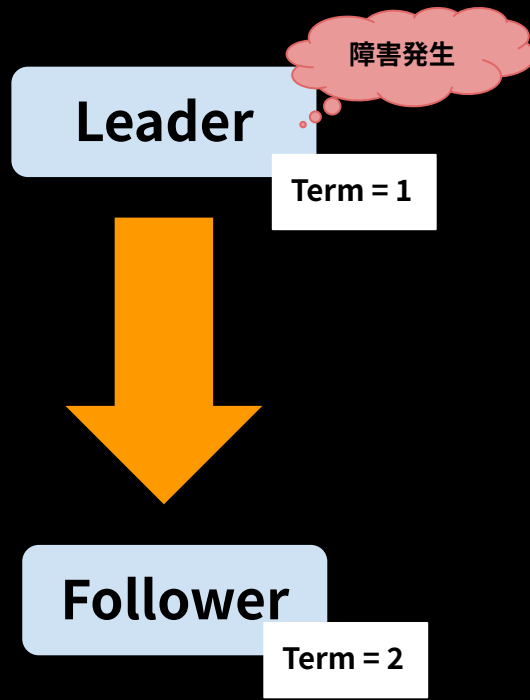
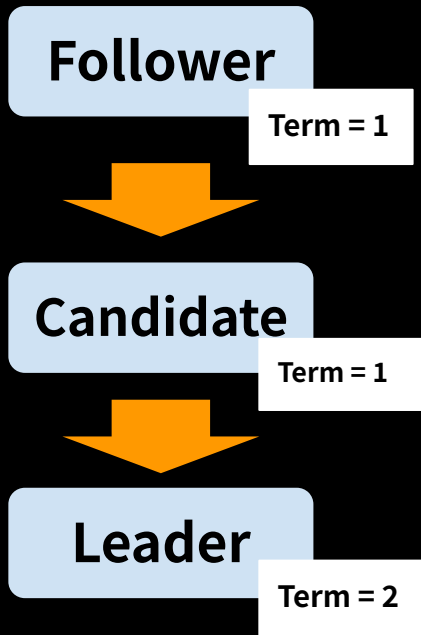
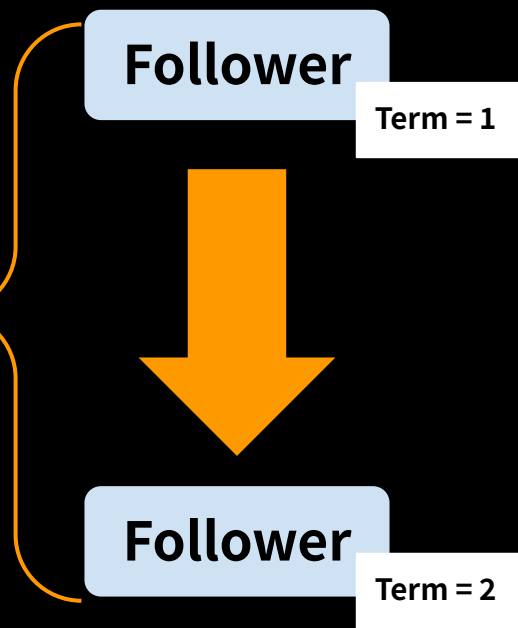


Raftによるログレプリケーションの動き



Raftによる選挙の動作

必ず最新のデータを持っているノードがLeaderに任命



03 リアルな課題と運用ノウハウ

TiDBのテーブル構造

※実際のKeyにはTableの情報も含まれます

TABLE ※デフォルト：CLUSTERD

```
CREATE TABLE t (
  `id` INT PRIMARY KEY /*T![clustered_index] CLUSTERED*/,
  `name` VARCHAR(255) );
```

(Key)

-

(Value)

Primary Key

-

row data

Key (PK)
1001
1002
1003

Value (row data)
やまだ
いとう
さとう

- 1レコードを1つのKey-Valueとして保存
- Key：PK - Value：その他のColumn

INDEX

```
CREATE TABLE t(
  `id` INT PRIMARY KEY /*T![clustered_index] CLUSTERED */,
  `name` VARCHAR(255),
  INDEX id_name (`name`));
```

(Key)

-

(Value)

Index

-

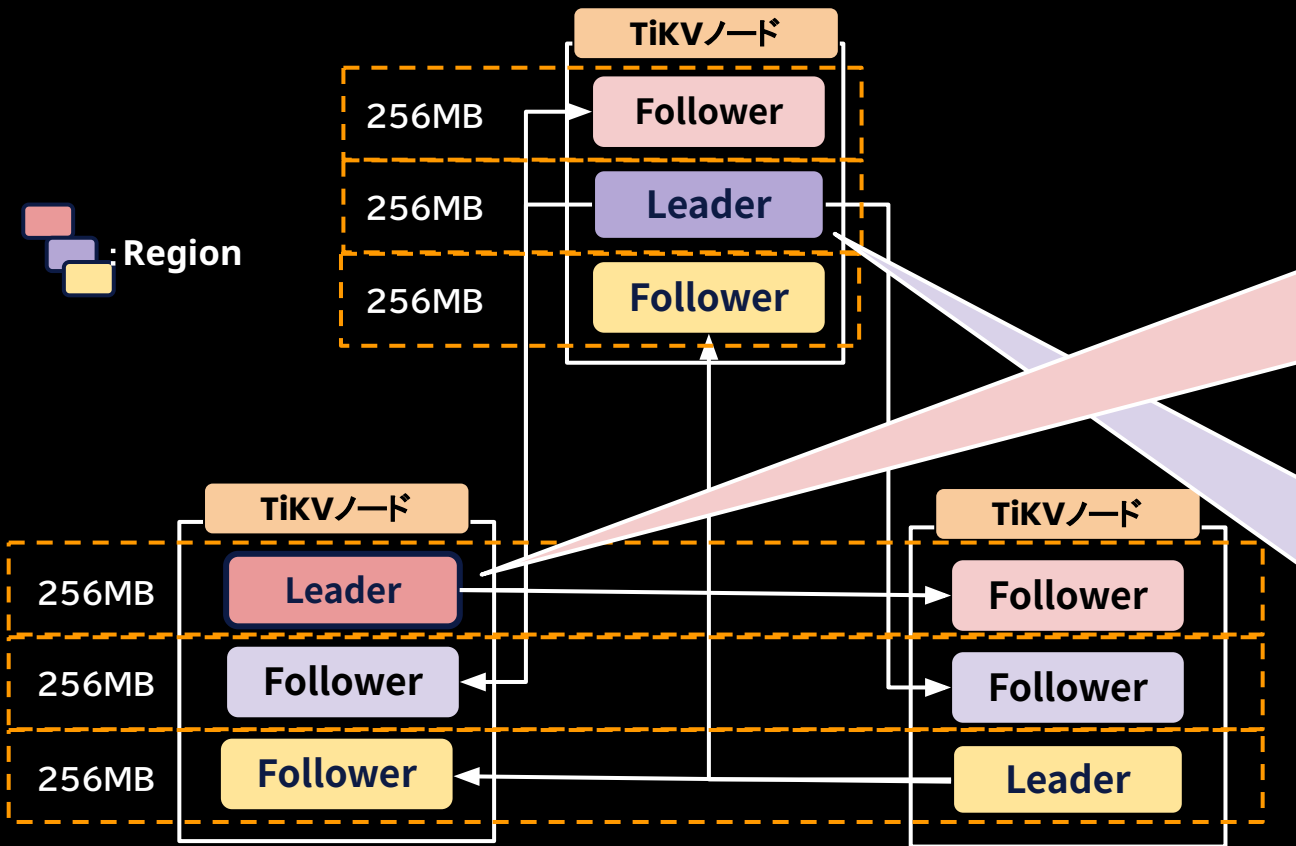
Primary Key

Key
やまだ
いとう
さとう

Value (row data)
1001
1002
1003

- Key：Index / Value：PK
- Index Lookup => 2ホップでアクセス
- ※①Index KV → ②Table (Row Data) KVの順

TiDBにおける自動シャーディングロジック



例：ユーザーテーブル

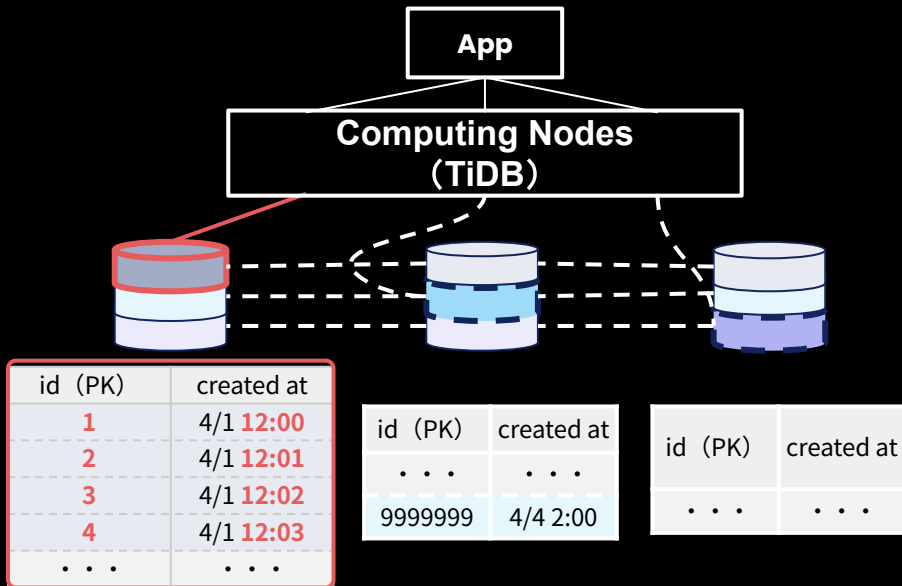
256MB			
id	名前	住所	電話番号
1	佐藤	北海道	1234
2	鈴木	青森	1235
3	高橋	秋田	1236
256MB			
4	田中	岩手	1237
5	伊藤	山形	1238
6	渡辺	茨城	1239

Writeの観点：PKを散らす

PKが連番

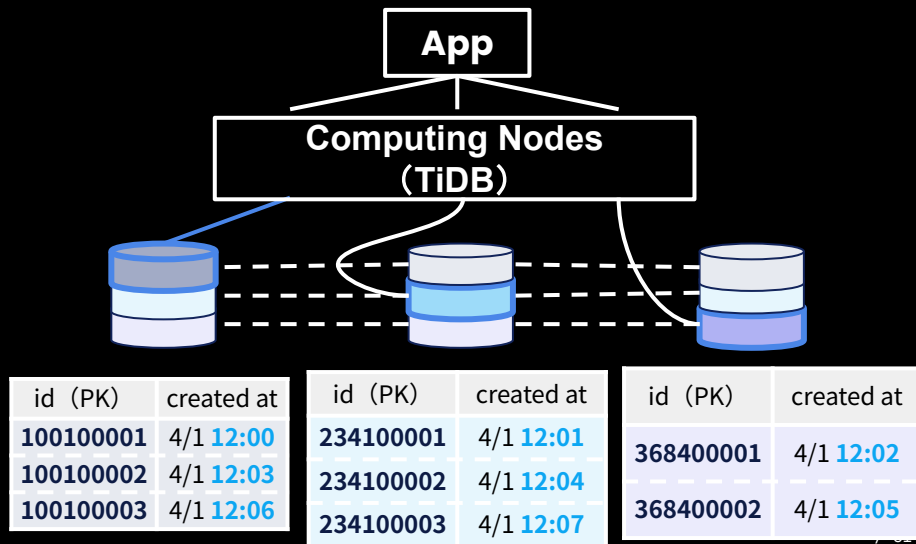
多数の近い値を主キーに挿入すると、
書き込みホットスポットを誘発

ex. Auto Incrementテーブルに大量INSERT



PKがランダム ※バラつきあり

主キーにはバラつきのある値を利用
-> 負荷もバラけるためHotspot回避
※Auto RandomやUUIDなど



Readの観点：PKの工夫



```
SELECT * FROM `user_action_logs`  
WHERE `user_id` = ? AND `created_at` >= ?  
ORDER BY `created_at` DESC, `id` DESC LIMIT ?
```

サロゲートキー + Index

```
CREATE TABLE `user_action_logs` (  
  `id` varchar(255) NOT NULL,  
  `user_id` varchar(255) NOT NULL,  
  `log_data` json DEFAULT NULL,  
  `created_at` timestamp NOT NULL,  
  `updated_at` timestamp NULL DEFAULT NULL,  
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */,  
  KEY `user_action_logs_user_id_created_at_index` (`user_id`,`created_at`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

id (PK)	user_id	created_at
asdijasoja	100001	8/19 12:00
zmpkogkp	100001	8/20 22:00
haoahapji	100001	8/21 5:00

TiDB



CPU負荷高め

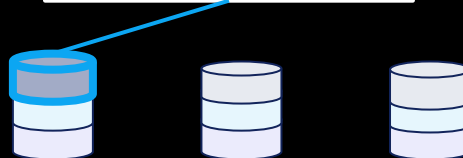
- PKがサロゲートキー (id) のため、取得したいデータが各TiKVノードに散らばっている
- Indexからのアクセスのため2ホップ

PK

```
CREATE TABLE `user_action_logs` (  
  `id` varchar(255) NOT NULL,  
  `user_id` varchar(255) NOT NULL,  
  `log_data` json DEFAULT NULL,  
  `created_at` timestamp NOT NULL,  
  `updated_at` timestamp NULL DEFAULT NULL,  
  PRIMARY KEY (`user_id`,`created_at`,`id`) /*T![clustered_index] CLUSTERED */,  
  UNIQUE KEY `unique_idx_id` (`id`),  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

user_id (PK)	created_at (PK)	id (PK)
100001	8/19 12:00	asdijasoja
100001	8/20 22:00	zmpkogkp
100001	8/21 5:00	haoahapji

TiDB



取得したいデータを特定のTiKVノードに収めることができるので効率が良い

=> **SLELECT**の条件に即してPKを設計しておくことがポイント
※RDBMSと同じ

Readの観点：Covering Index ※PK除いた全カラムのIndex

テーブルのKV

Primary Key - row data

id (PK)	user	department	created at
1	やまだ	AAA	4/1 12:00
2	いとう	AAA	4/1 22:00
3	さとう	BBB	4/2 5:00
4	いしだ	BBB	4/2 13:00

```
SELECT * FROM `user_action_logs`
WHERE `user_id` = ? AND `created_at` >= ?
ORDER BY `created_at` DESC, `id` DESC LIMIT ?
```

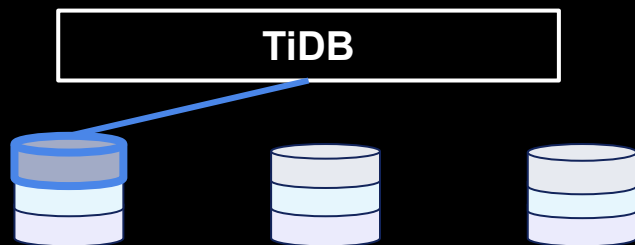
```
CREATE TABLE `user_action_logs` (
  `id` varchar(255) NOT NULL,
  `user_id` varchar(255) NOT NULL,
  `log_data` json DEFAULT NULL,
  `created_at` timestamp NOT NULL,
  `updated_at` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */,
  INDEX idx_covering(`user_id`,`created_at`,`updated_at`,`log_data`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

IndexのKV → こっちだけを見る

Index - Primary Key

user	department	created at
やまだ	AAA	4/1 12:00
いとう	AAA	4/1 22:00
さとう	BBB	4/2 5:00
いしだ	BBB	4/2 13:00

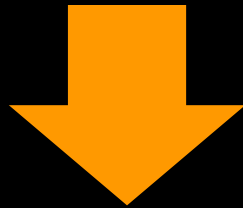
id (PK)
1
2
3
4



取得したいデータを特定のTiKVノードに格納
※ストレージが2倍になる
(テーブルとIndexのKV)

①: 帯域（数百万QPS）やレイテンシに対する設定

例えば、ワークロードとしてRead heavyだった時
TiDBの性能を発揮するパラメータとは？



2024年、TiKVのCPU使用率が高くなると、
Write latencyが上がる課題があった

v8.0でstore-pool-ioパラメータのデフォルト値を変更: 0->1
=> Raft I/O タスクを処理するスレッドの許容数

②: 更新ヘビーなワークロードが起こす問題

- TiKVはLSMTree(追記型)
- Write(update,delete)は追記型で保存
- => ある程度の頻度(compaction)で削除

履歴(MVCC)データが多くなる
=> 履歴データを含めた検索



- ・TiKV CPU使用率の増加
- ・レイテンシーの増加

TiDB側としての改善

①: GCで削除するの頻度を増やす
(parameter :compaction-filter->false)

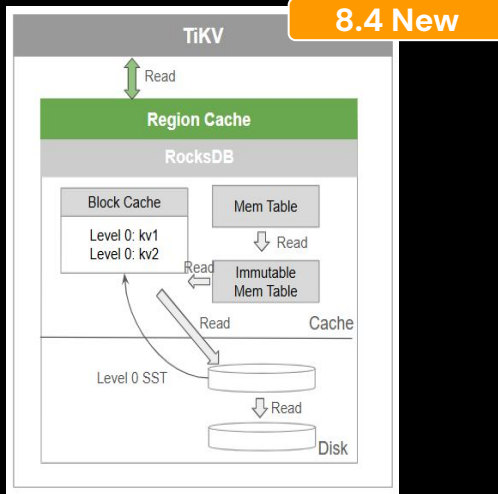
GC in Compaction Filter

Based on the DISTRIBUTED GC mode, the mechanism of GC in Compaction Filter uses the compaction process of RocksDB, instead of a separate GC worker thread, to run GC. This new GC mechanism helps to avoid extra disk read caused by GC. Also, after clearing the obsolete data, it avoids a large number of left tombstone marks which degrade the sequential scan performance.

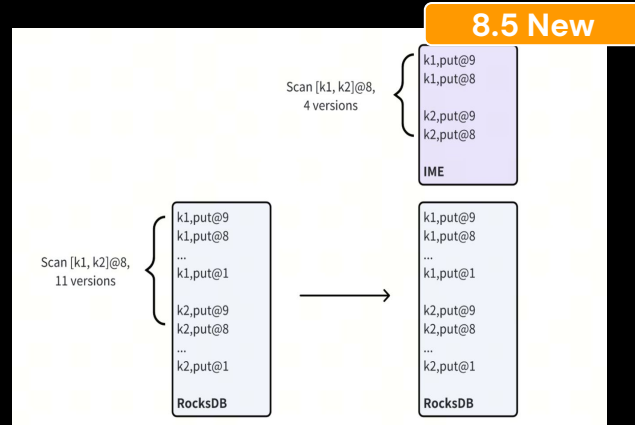
The following example shows how to enable the mechanism in the TiKV configuration file:

```
[gc]
enable-compaction-filter = true
```

②: TiKVの中にキャッシュを作成
最新のデータのみ読む機能
(in-memory cache)



③: TiKV内にMVCC用の
インメモリエンジンを搭載
最新のMVCCバージョンを
メモリにキャッシュ
(in-memory-engine)



③: 多テーブルが引き起こす問題

B2B2C/つぎ足すシステムはテーブルが多くなる傾向がある



統計情報のロードに時間がかかる
(起動が遅い、クエリが非効率に)

Information Schemaのレスポンス低下
(show~のせいでmigrationが遅い。)

TiDB側としての改善

1年以上かけて改善し続けています (統計情報のロード性能など)

Companies that use TiDB to run multi-tenant or SaaS applications often need to store a large number of tables. In v8.5.0, TiDB significantly enhances the stability of large-scale clusters.

`tidb_pre_split_regions`

Improve the stability of large-scale clusters

Newly added

Before v8.4.0, setting the default number of row split slices for newly created tables required declaring `PRE_SPLIT_REGIONS` in each `CREATE TABLE` SQL statement, which is complicated once a large number of tables need to be similarly

- Improve the query performance of some system tables [#50305 @tangenta](#)

In previous versions, querying system tables has poor performance when the cluster size becomes large and there are a large number of tables.

In v8.0.0, query performance is optimized for the following four system tables:

- `INFORMATION_SCHEMA`
- `INFORMATION_SCHEMA`
- `INFORMATION_SCHEMA`
- `INFORMATION_SCHEMA`

Improve statistics loading efficiency by up to 10 times

For clusters with a large number of tables and partitions, such as SaaS or PaaS services, improvement in statistics loading efficiency can solve the problem of slow startup of TiDB instances, and increase the success rate of dynamic loading of statistics. This improvement reduces performance rollbacks caused by statistics loading failures and improves cluster stability.

`tidb_shard_row_id_bits`

8.5

8.4

8.3

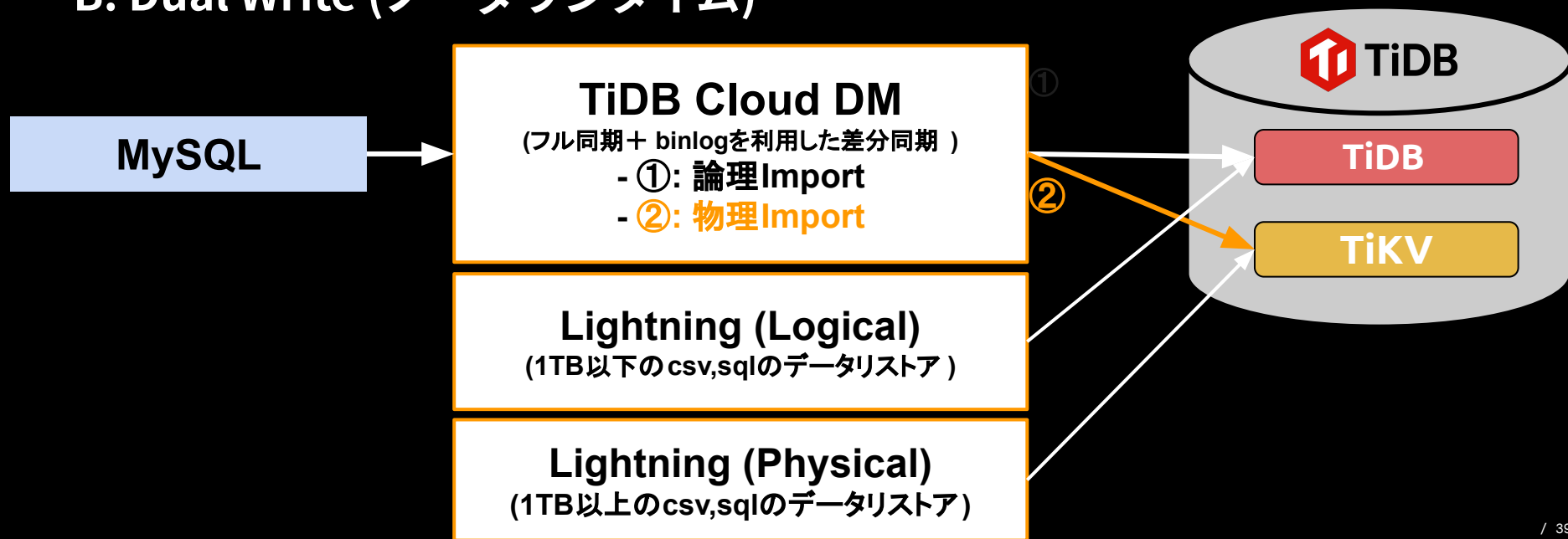
8.2

④: 用途に応じた移行方法の複雑化

様々な移行方法例

A. TiDBのエコシステムツールを利用

B. Dual Write (ノーダウンタイム)



移行時のテクニック

柔軟なスケーリングを考慮したキャパシティプラン

移行直後

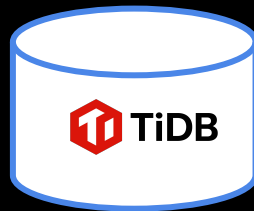


CPU増・スロークエリ発生

- Indexが足りない
- クエリが非効率

チューニング

移行後



オンライン
スケールイン/ダウンで
最適なサイジングへ



Thank you !

Koitabashi Yoshitaka

X yoshii0110

