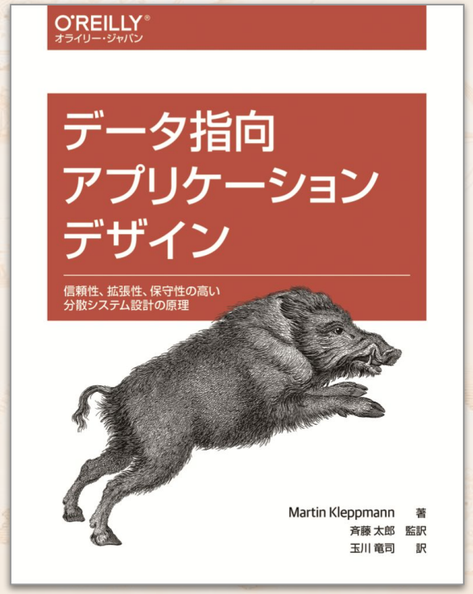
30分でわかるデータ指向アプリケーションデザイン

Taro L. Saito

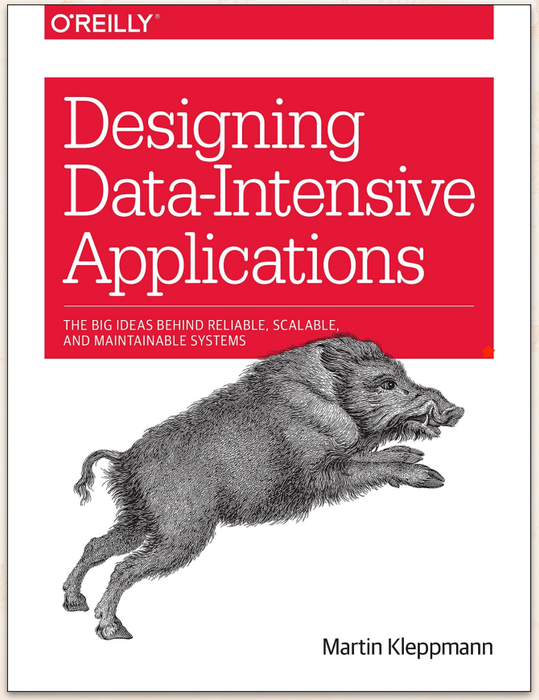
Data Engineering Study #18

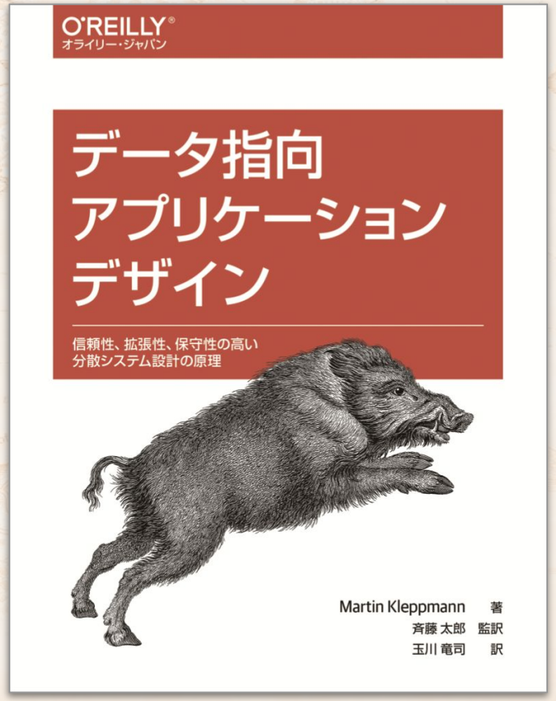
February 15, 2023



データ指向アプリケーションデザインとは

- 原著：Designing Data-Intensive Applications
  - 「データ指向」とは、この本のために生み出された新しい訳語
  - Intensiveの従来の訳語: CPU, Memory-Intensive (CPU、メモリ集中、特化型)
  - では、データ集中、データ特化型？（イマイチ！）
- オブジェクト指向 (object-oriented) : オブジェクトを中心にプログラムを設計する
- **データ指向：「データの量、複雑さ、変化の速度」を中心に考えるデザイン**

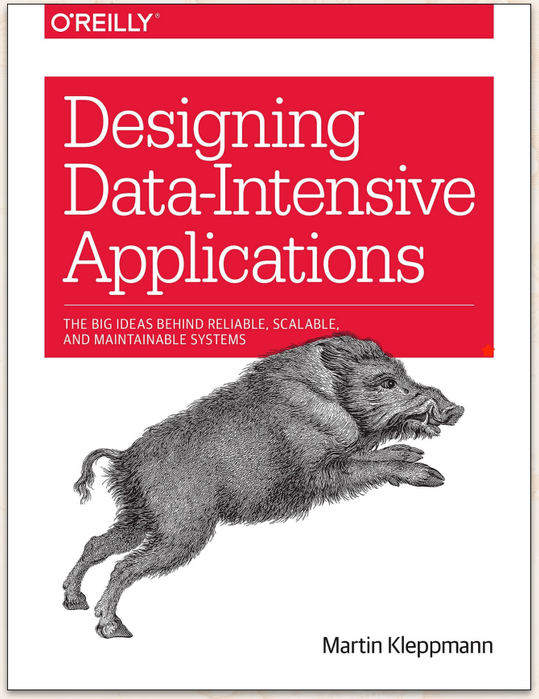


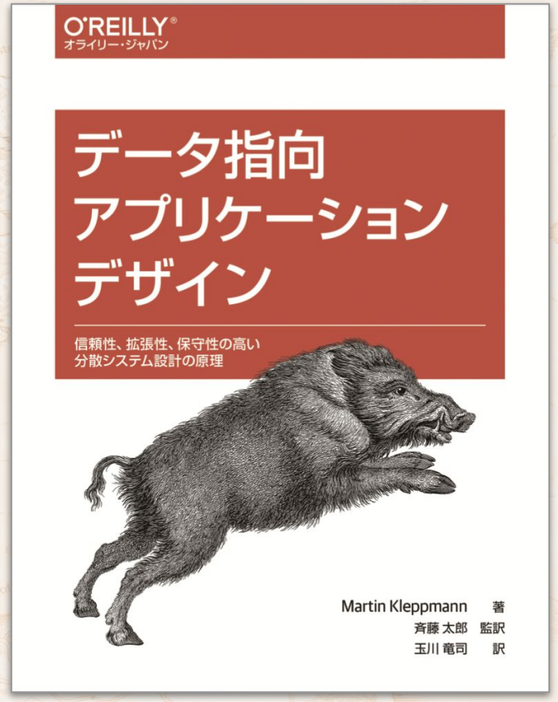


「データ指向アプリケーションデザイン」から5年

Advanced Topic

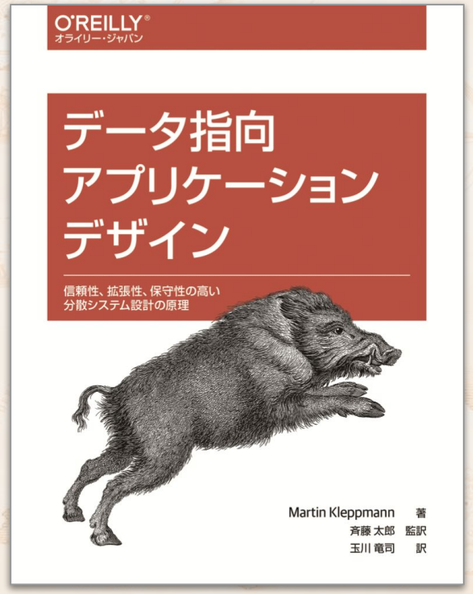
- 出版年：原著 (2017)、邦訳版 (2019)
  - ここ5年の間にも新しい技術、論文、製品が次々に生まれている
- 新しい話題、発展的な話題、本書にない補足情報を扱う際にはスライドにリボンをつける
  - 新しい技術を理解する際にも、「データ指向アプリケーションデザイン」の基礎が役立つ





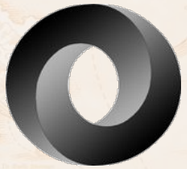
「データ指向」を極める

「データの量・複雑さ・変化の速度」に対応できるシステム設計



まずはデータを作るところから

- **データ量が少ない場合**
  - データの表現、生成しやすさが重視される



- テキストデータフォーマット
  - CSV、JSON、XMLなど。テキストエディタでも記述できる
  - 正確なデータ型をコンパクトに表現したい場合に若干困る
- バイナリデータフォーマット
  - MessagePack
  - Thrift (現在ではApache版と、Facebook版が分離)
  - ProtocolBuffers (Google)
  - Apache Avro

- データとして完結している**自己記述型 (self-describing)** データフォーマット
  - 例外：ProtoBufはデータ自体にスキーマを含まない(self-describingではない)ので、かなり節約指向

データ量が多い場合

- データ基盤では省メモリ、圧縮しやすさ、ストレージへの格納しやすさが重視される
- テーブルフォーマット
  - スキーマ（各カラムの名前、型）とレコード表現を分離
    - カラム名、型の情報を繰り返して出力しない分、コンパクト
- 行指向(row-oriented) フォーマット
  - RDBMSでよく使われる
  - 一行（1 record）ずつ表現
  - レコード単位で更新しやすい
- 列指向(column-oriented)フォーマット
  - 列（カラム）
  - 同じ型のデータを集めると圧縮しやすい
  - 分析クエリ向き
    - キャッシュに乗りやすく高速
    - SIMD演算も活用しやすい
  - 一方、単一レコードの更新は大変

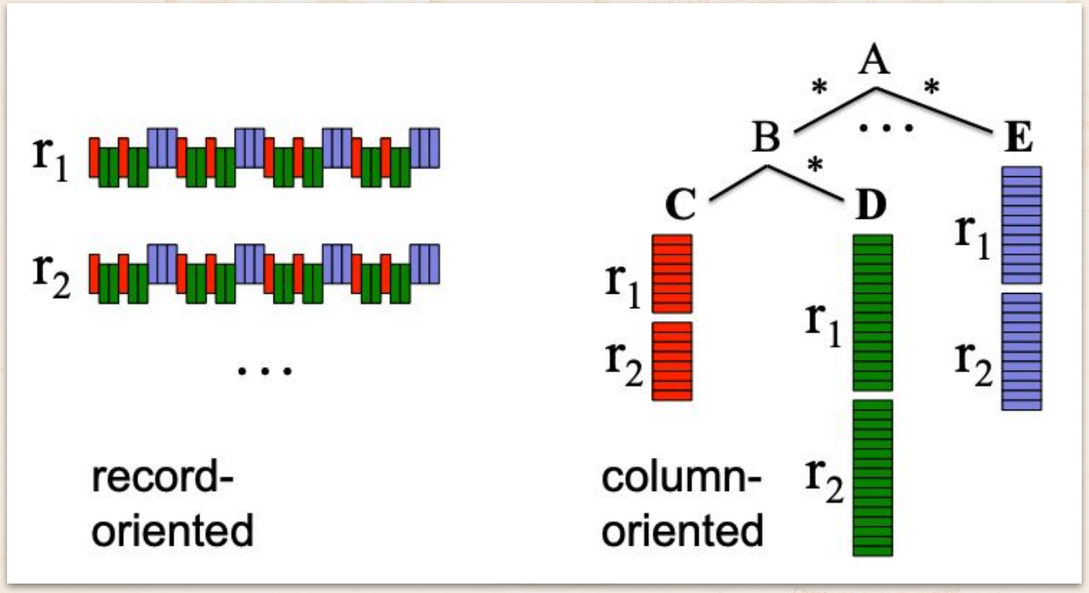
id	name	email	...
1	xxx	xxx@xxx.xxxx	
2	yyy	yyy@yy.yyy	
3	zzz	zzz@zzzz.zzz	

id	name	email	...
1	xxx	xxx@xxx.xxxx	
2	yyy	yyy@yy.yyy	
3	zzz	zzz@zzzz.zzz	

列指向データフォーマット (Columnar Format)

Advanced Topic

- 例 : Apache Parquet
  - Google Dremel (VLDB2010) のアイデアがベース
  - ネストして繰り返しのあるデータでも、同じパスごとに分解、圧縮する
    - Spark、DuckDBなど、エコシステムで広くサポートされるように
- クラウド上のストレージ（AWS S3など）で扱いやすくするための工夫
  - S3:スループットは高いが、ランダムアクセスのlatencyが大きい (10ms - 数百ms)
  - 1ファイル内でページ分割、フッターに各ページへのインデックスを格納



DocId

value	r	d
10	0	0
20	0	0

Name.Url

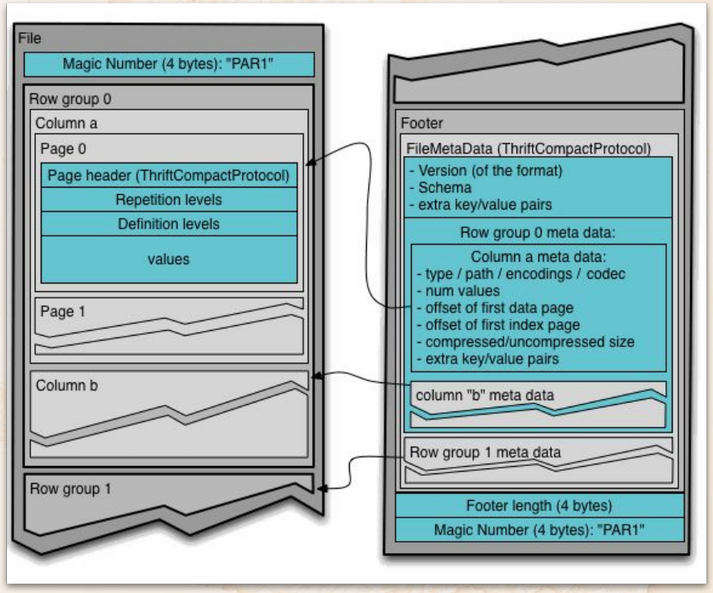
value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1



データが変化する速度は？

- 変化しない場合：**不変データ（immutable data）**
  - 例：蓄積されるログデータ。列指向フォーマットが使いやすい
- 読み込み特化型 **(read-intensive)**
  - データへの高速なアクセス、分析の速さが問われる
- 読み込み特化型の高速化手法：**データを分散する**
  - パーティショニング（シャーディングとも言われる）
    - データを時間範囲（1時間、1日など）、あるいはキー範囲などで分割する
    - 必要なデータのみにアクセスしたり、別領域のデータへの並列アクセスを可能に
  - レプリケーション
    - 同じデータを複数箇所（マシン、ディスク）に配備する
    - 並列アクセスしやすくし、耐障害性を持たせる
  - 実体化ビュー (Materialized View)
    - 一度計算したクエリ結果を保存し、再利用できるようにする

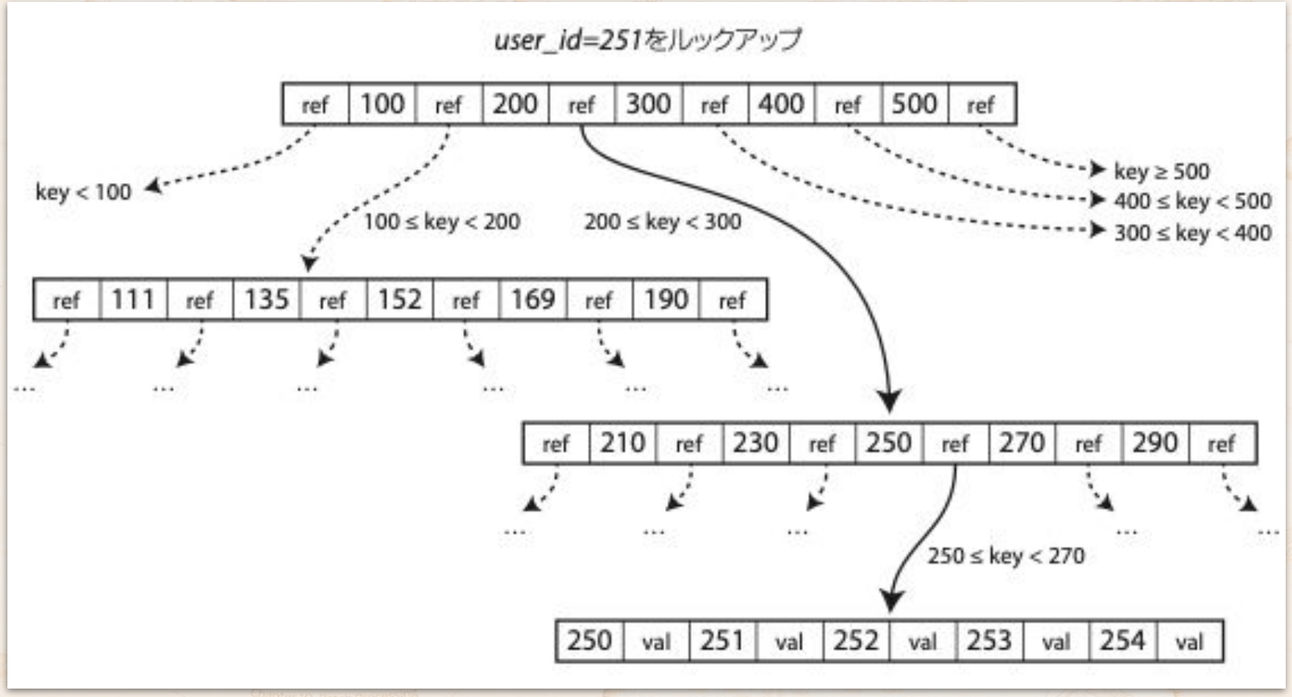
データが頻繁に更新される場合

- 書き込み特化型 (**write-intensive**)
  - RDBMSが強い分野
    - レコード単位での更新
    - トランザクション処理

- 更新対象をどう素早く見つける？
  - インデックス：ディスク上のデータを高速に見つけるためのデータ構造
    - B-Tree, Hash index、SSTable (Sorted String Table)

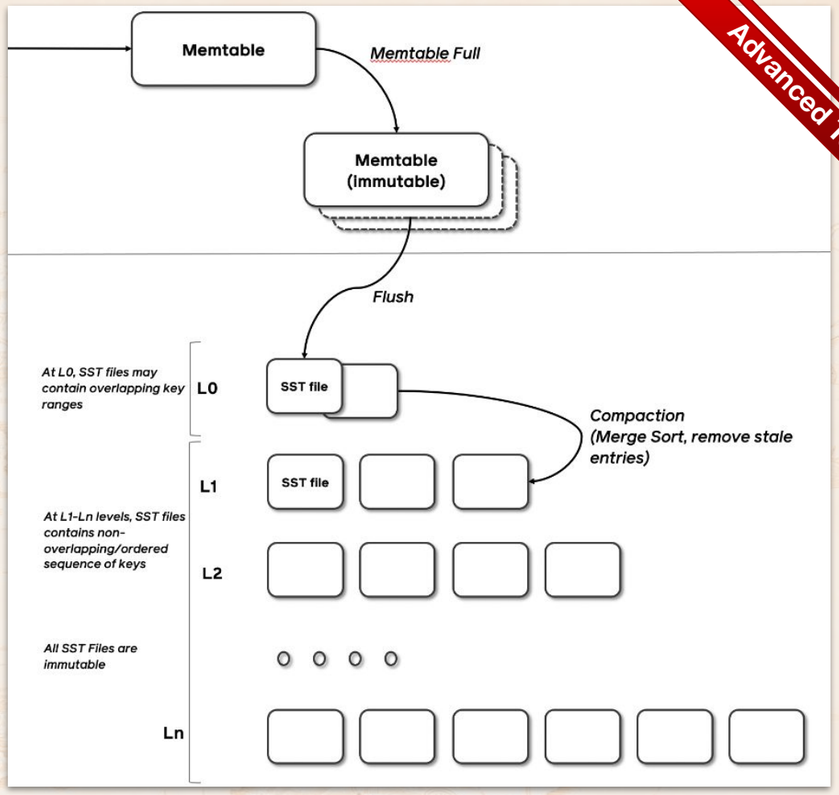
ディスク上のインデックス構造：B-Tree

- ディスク上のページに、次のページへのルックアップテーブルを格納
  - RDBMSでは、基本CREATE INDEX命令で作られる
- 高い更新性能：空き領域にデータを挿入しやすい
- ページ数が増えても木の高さがあまり増えないので、ディスクへのランダムアクセスを減らせる
  - 例：各ノードに100個のエントリがある場合。1億エントリがあっても、木の高さは、log_100(100,000,000) = 4にしかならない。つまり4回のlookupで目的のデータページが見つかる

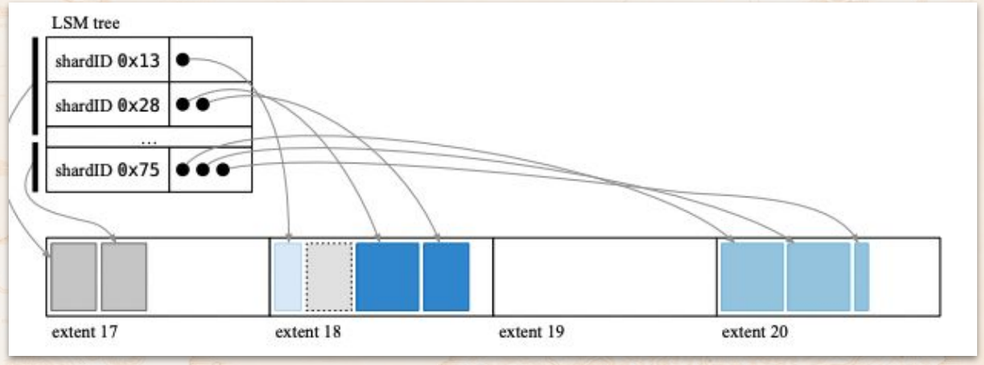


発展：Log-Structured Merge (LSM) Tree

- 更新がさらに頻繁な場合にどうするか？
- レコードの追加、コンパクション操作を分離する
  - メモリ：ランダムアクセスに強い構造
    - 例：更新されるたびにメモリ上でソートしておく
  - L0, L1, ...：ディスク、外部ストレージなど
    - 定期的にソート済みのデータと合成(merge sort)して次のレイヤーに保存する
- 性能特性
  - 更新時のランダムアクセスを減らせる
    - SSD内部でも、一度キャッシュメモリでランダムアクセスを吸収してから、シーケンシャルアクセスに変換し、セルの寿命を伸ばしている
  - 各種オーバーヘッド
    - 読み込み時に各レイヤーにアクセスする必要がある
    - 書き込みの増幅（Write Amplification）が生じる
- 関連
  - LevelDB、RocksDB(C++)、AWS S3(Rustで実装）など



RocksDB Overview



Amazon S3 Data Structure (SOSP 21)

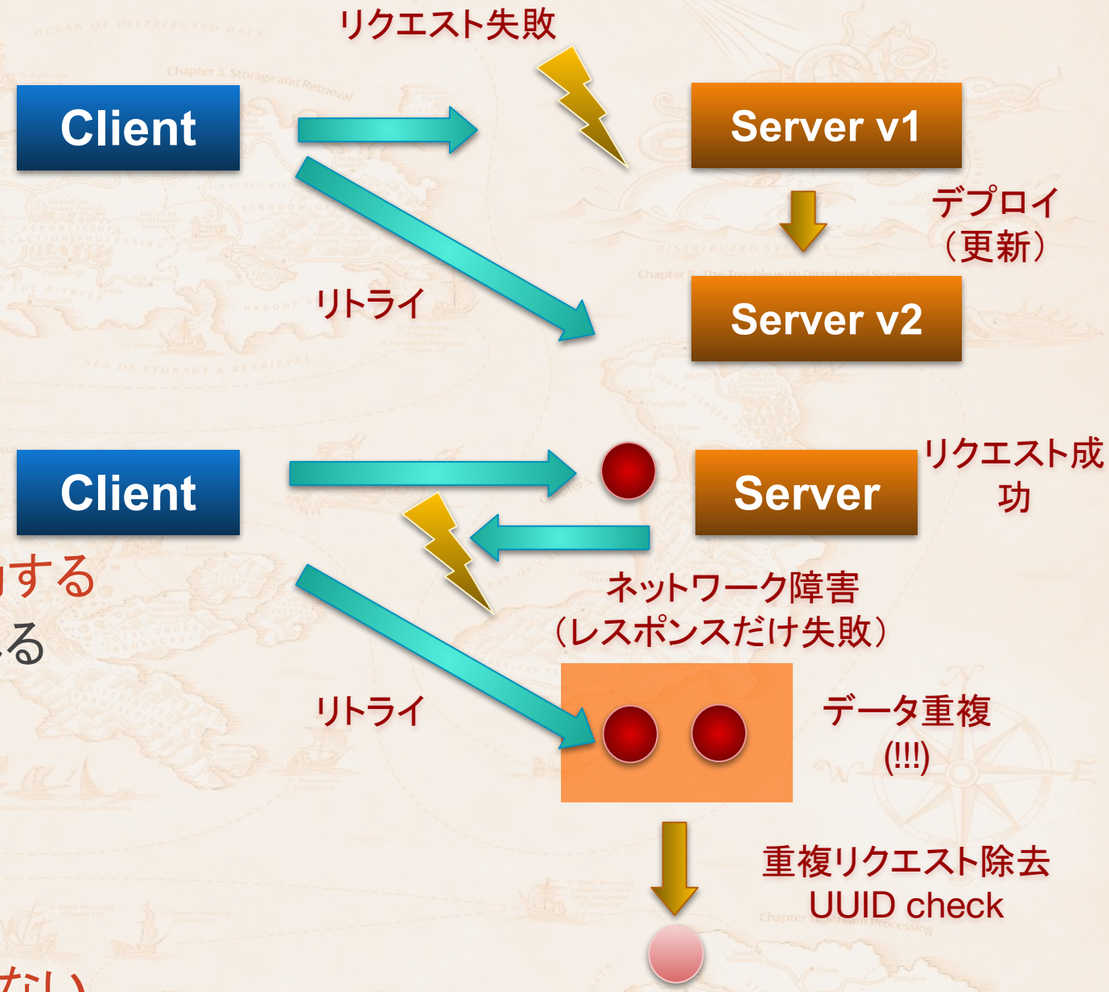
分散データでの新たなチャレンジ

- 1箇所にデータがある場合と比較して、**考えるべき問題の複雑さが数段上がる**

- そもそもノード間でデータを送受信するために、**分散システムが必須**
  - 分散ステートマシン、つまり各ノードが同じ状態になる必要（membership）がある
  - サーバー・クライアント間でのデータ通信 (HTTP上で実装されることが多い)
    - RESTプロトコル、RPC (Remote Procedure Call)
- 複数箇所で頻繁に更新される場合
  - 同期（ロックの取得、タイムアウト、トランザクション、ログのリプレイ）
  - どの状態が正解かを決めるための合意（consensus）プロトコル
    - 多数決 (Quorum) で決める：Paxos、2PCなど
    - あるいは、そもそも調整を避ける（coordination avoidance)
- 障害対応
  - エラーハンドリング、リトライ、冪等性 (idempotency)、障害からの復旧・リカバリ
  - ノードが嘘を付く（間違ったデータを返す）ビザンチン障害と呼ばれる

24時間365日動き続けるサービスの設計

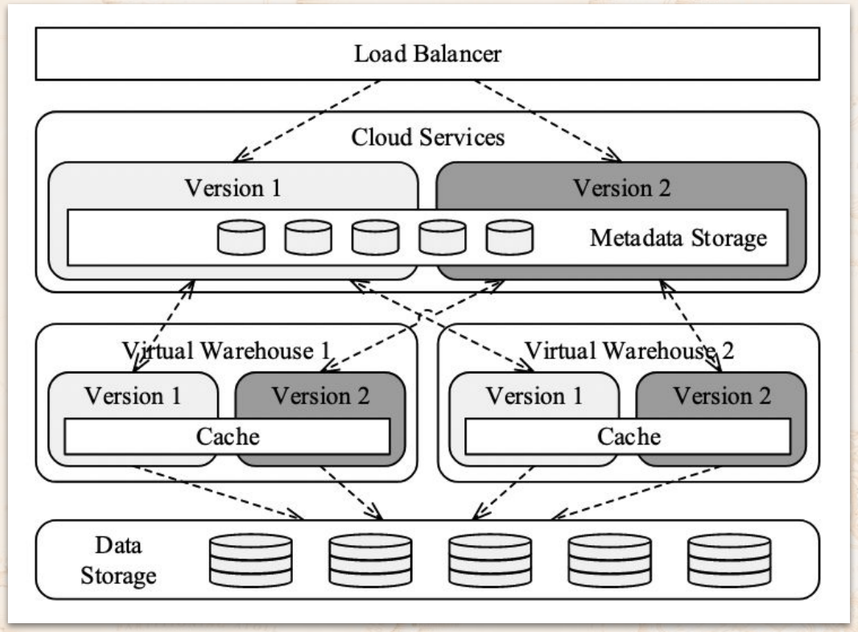
- 稼働率 (Availability)
  - 許容されるダウンタイム
    - 99.9% availability　8.7時間/年
    - 99.99% availability　53分/年
    - 99.999% availability 5分間/年
- アプリケーションのデプロイ時
  - サービスが動き続ける必要がある
    - 複数バージョンのアプリケーションが同時稼働する
  - デプロイ中、あるいはクラッシュ時に通信が遮断される
  - クライアント側でのリトライ（再実行）処理
    - サーバー側の稼働率の要件を緩められる
- 冪等性 (Idempotency)
  - リトライが起こる前提で設計する
    - 同じ操作が繰り返されても、データを重複させない
  - リクエストにUUIDなどをつけて重複をチェックする



Compute - Storageの分離

Advanced Topic

- 常にデプロイし続けられるよう、クエリの実行（Compute) とストレージ(Storage)の分離するのが近年の標準
  - クエリエンジンとストレージを別々にスケールしたり、ローリングアップグレードができる
- 例：Snowflake、Amazon Redshiftなど



Snowflake (SIGMOD 2016)

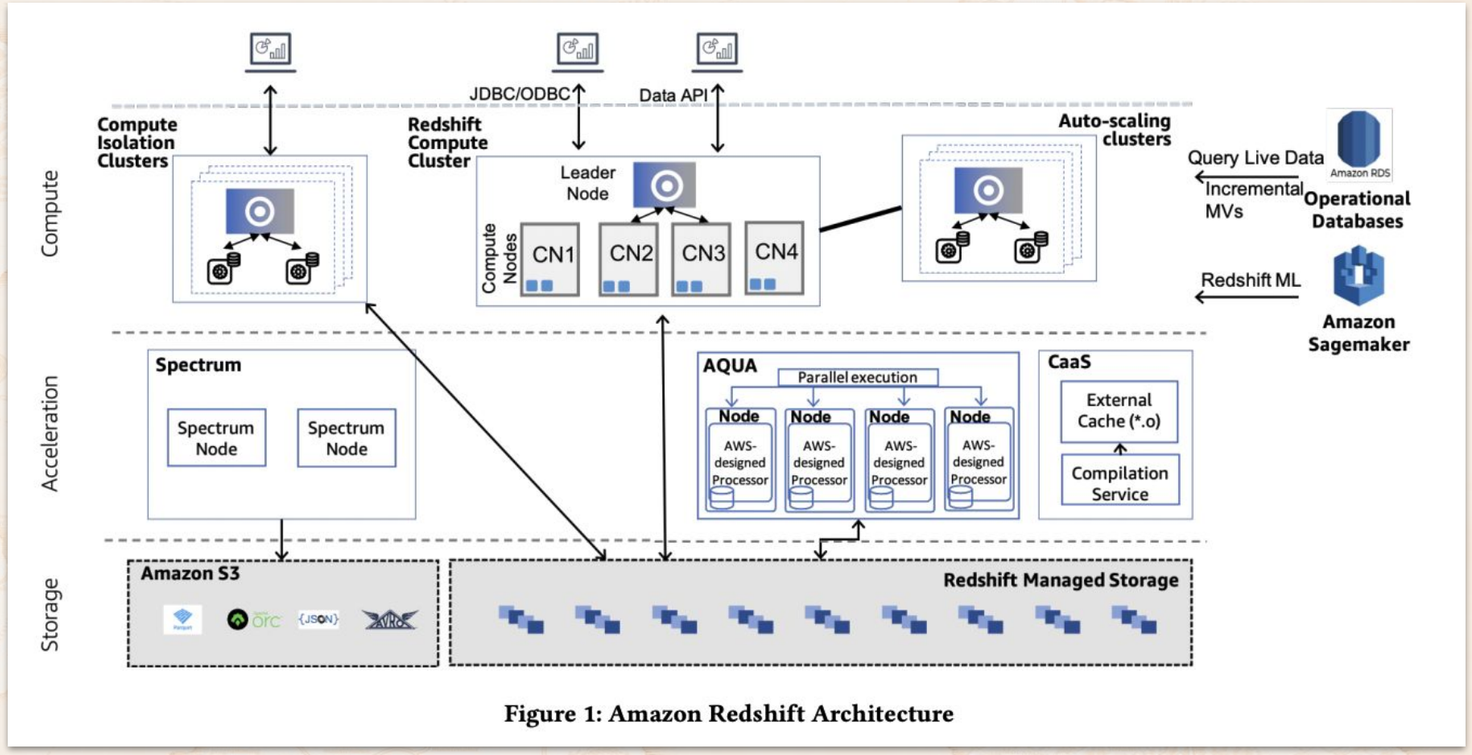
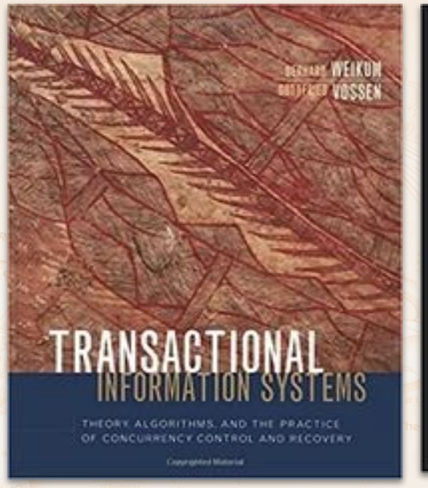
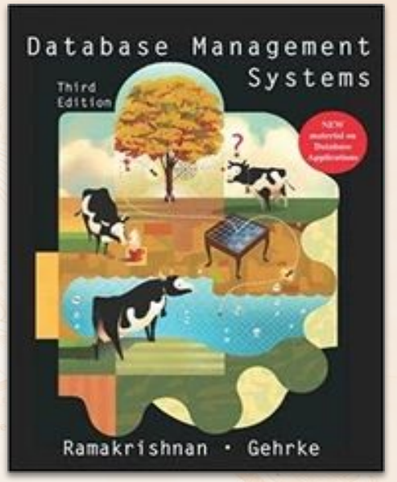


Figure 1: Amazon Redshift Architecture

Amazon Redshift (SIGMOD 2022)

トランザクション処理

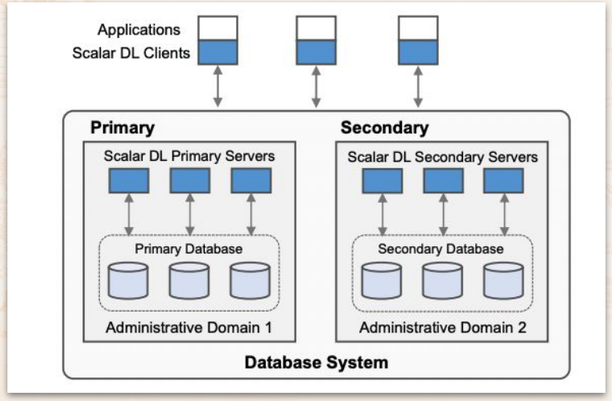




- データベースの花形技術
- ACID (1983年に提唱されたキーワード)
  - Atomicity, Consistency, Isolation, Durability
  - 単にACIDと言う場合、ほぼマーケティング用語
    - 何がどう保護されるかは、実装の詳細まで見ないとわからない。実は専門家になるほど、ACIDの意味がわからなくなる
- 分離性（Isolation）の概念が最も重要
  - 直列化可能性 (Serializability) が基本
    - トランザクションが1つずつ実行されたのと変わらない状態を維持する。オーバーヘッドが大きいため現実的には使われにくいが、逐次実行しても性能が得られる場合も
  - 弱い分離性
    - Read-Committed、Snapshot Isolation、MVCC (Multi-version concurrency control)
  - 妥協しないアプローチ： Serializable Snapshot Isolation (SSI) (SIGMOD 2008 best paper)
  - トランザクション異常(anomaly)の種類
    - ファントム（まだ存在しないレコードに、どうロックをかけるか）
    - Write Skew (同じスナップショットを読んで、違う場所に書き込む）
  - Repeatable Read (という紛らわしい名前）の由来

そして、分散トランザクションの世界へ

- 複数ノード間、複数システム間でトランザクションを実装する
- 一貫性、合意、耐障害性
  - 線形化可能性 (linearizability)
    - レプリカが複数あっても、コピーが1つしかないように見せる。つまり、更新操作が終了した途端、全ノードが同じデータを見られるようにする
- 要素技術 (9章に詳しい)
  - リーダー選出 (leader election) 、サービスディスカバリ、メンバーシップ管理
    - これらが正しく実現、シングルリーダーレプリケーションなども正しく実装できない
- 関連：Zookeeper、etcd (Kubernetes内部で使われる)
- 異なった方向性の新しいアプローチ (2022~)
  - アプリケーション側のクライアントで分散トランザクション処理を吸収 (ScalarDB)

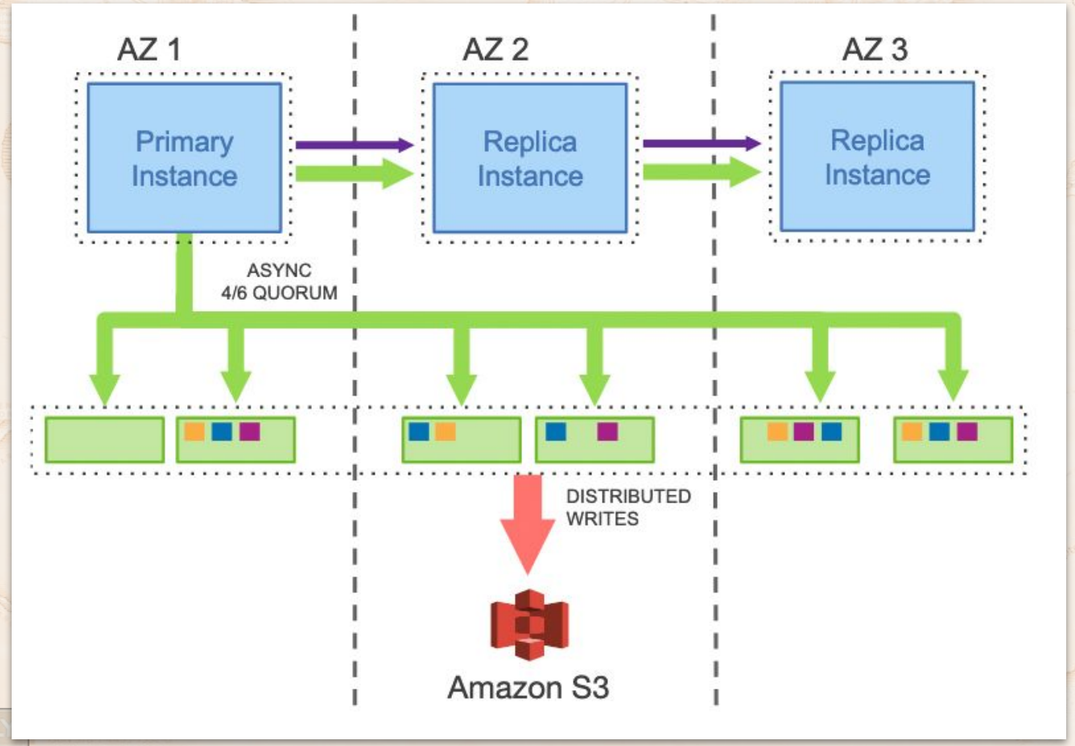


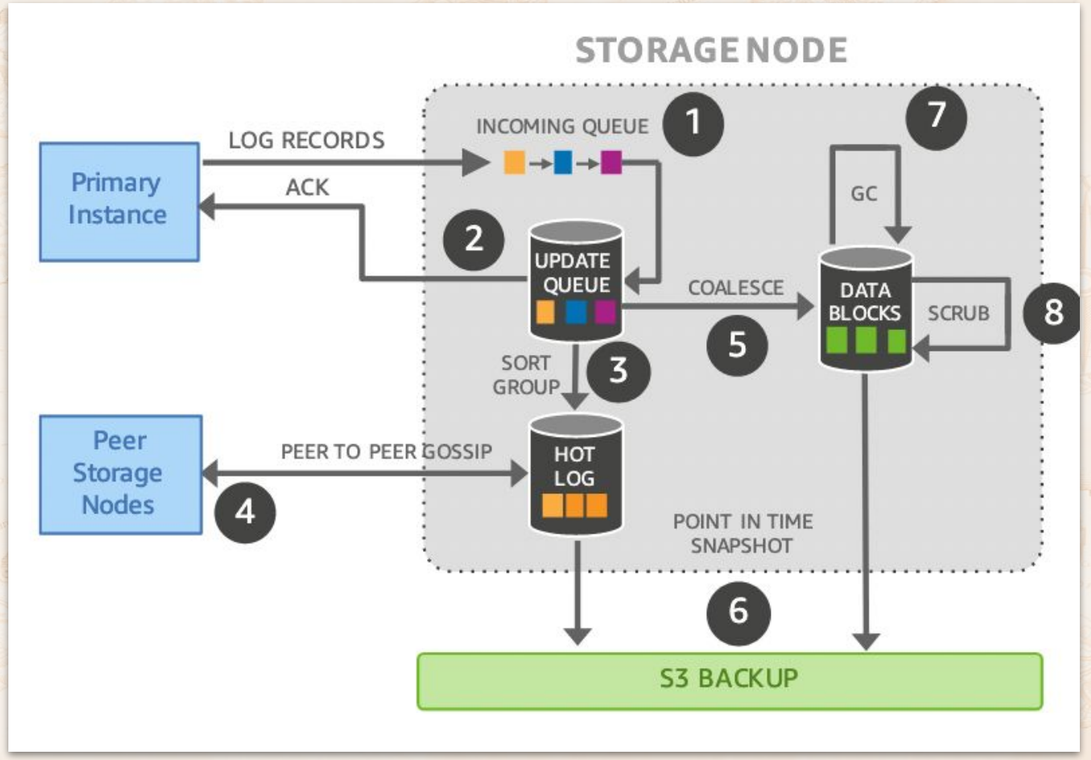
Scalar DL (VLDB 2022)

分散トランザクションの極み：Amazon Aurora (SIGMOD 2018)

Advanced Topic

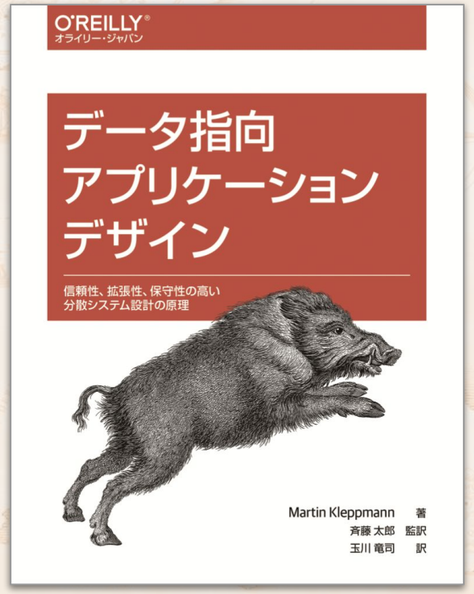
- 分散版MySQL：6つのレプリカを作成
  - 2 copy x 3 AZs. Availability Zone (AZ)が1つ落ちてもquorum(多数決)を取れる：4/6
- MySQLのデータベース更新操作を「ログの書き込み」だけに簡略化
  - ストレージノードがログを分配、各インスタンスでログをリプレイして同じデータを持つ
    - gossipプロトコル（ランダムにノードを選んで配信、同期システムなしに実装できる）
  - さらに、トランザクション処理も同期をなるべく取らずに処理（Coordination Avoidance)
    - 各ノードの返信を待つ2相コミット(2PC)を使わなくても良いよう、MySQLの裏側のストレージ全体を再設計





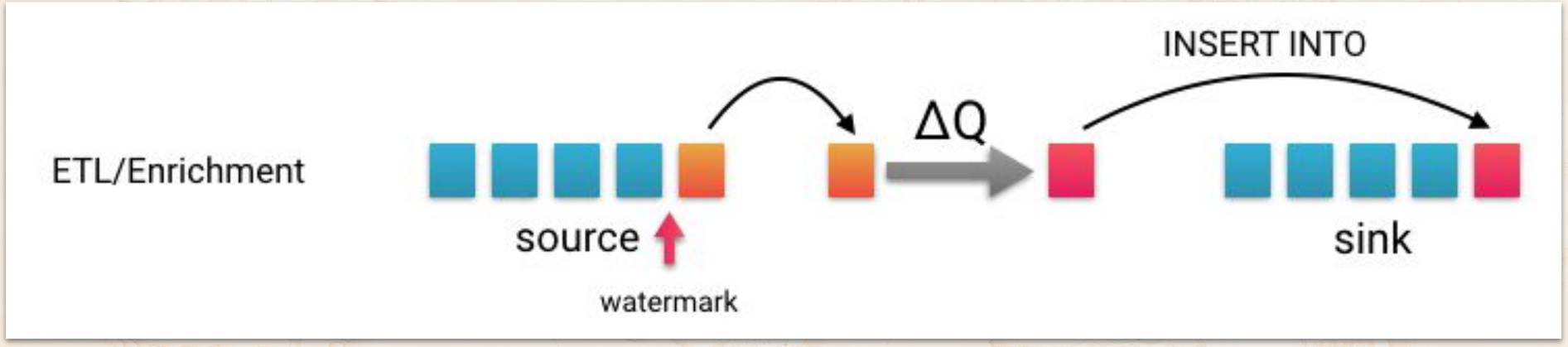
「データの複雑さ」の変遷

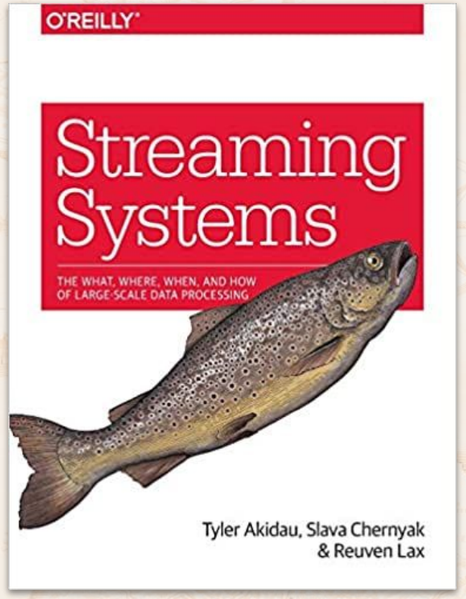
導出データ(derived data)の管理



データの複雑さ:「テーブル」の意味の変化

- 従来の「テーブル」の定義
  - RDBMSにある**最新のデータ** (snapshot)
- 現代での「テーブル」の意味
  - **時間とともに変化するデータ**、そこから**派生するデータ(derived data)**を表すように
  - 列指向クエリエンジンの発展により、分析クエリが容易になり導出データが大量に
    - Redshift, BigQuery, Trino, Spark, DuckDBなどのクエリエンジン、サービス
- バッチ処理、ストリーム処理
  - 近年では両者の区別は少なくなってきいる

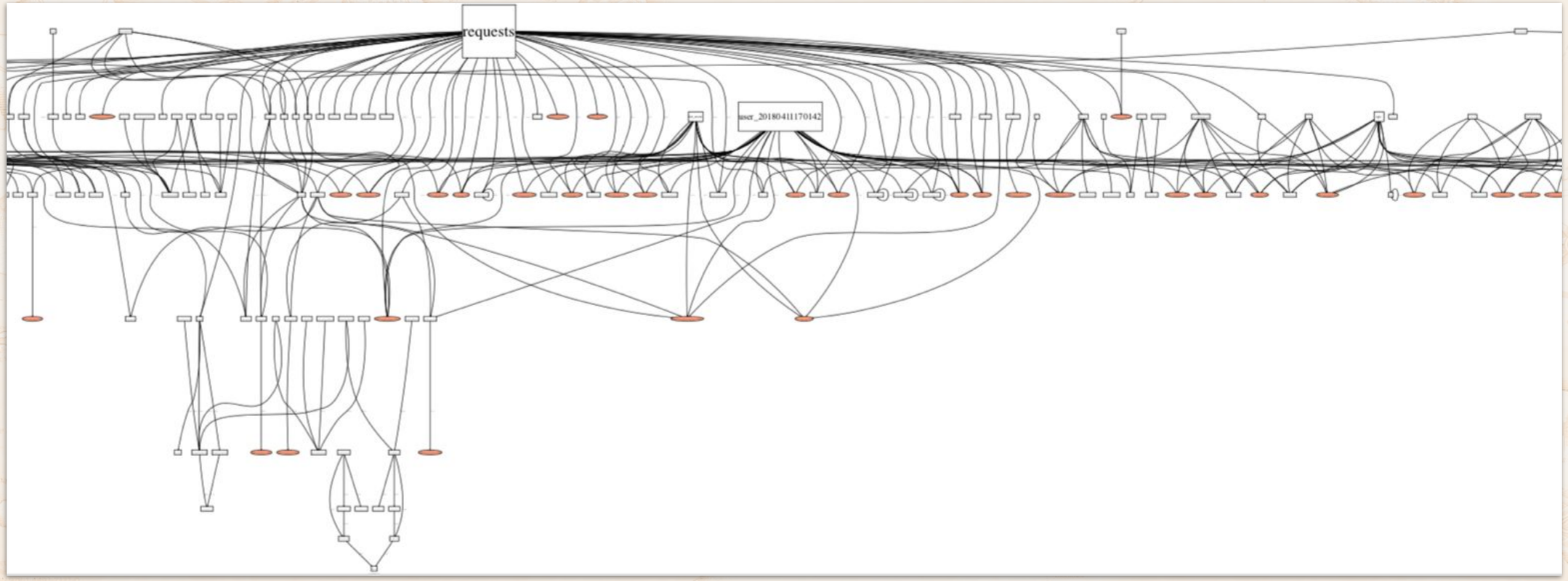




導出データ (Derived Data)

Advanced Topic

- 数千以上の導出データが生成される実例 （Treasure Data社）
- 例：クエリで生成されるデータの依存関係や、データの履歴を管理するには？
  - dbt: クエリの依存関係を記述できるSQL compiler
  - 新しいTable Format: Delta Lake、Iceberg、Apache Hudiなど。
    - テーブルの更新履歴の確認、スナップショットの管理(time travel)ができる

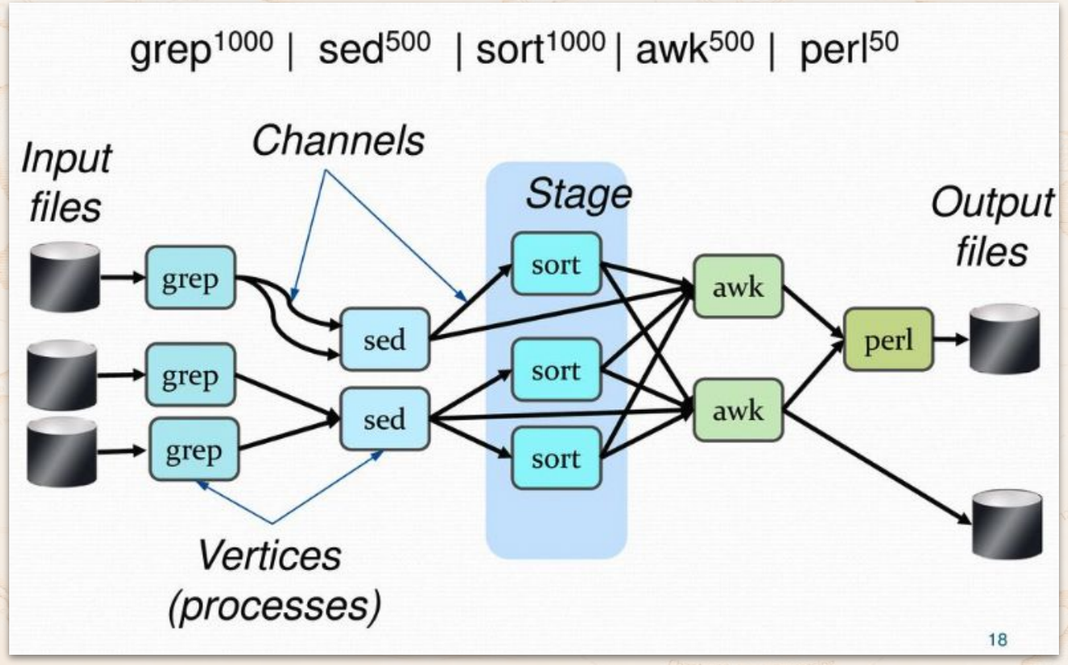


バッチ処理

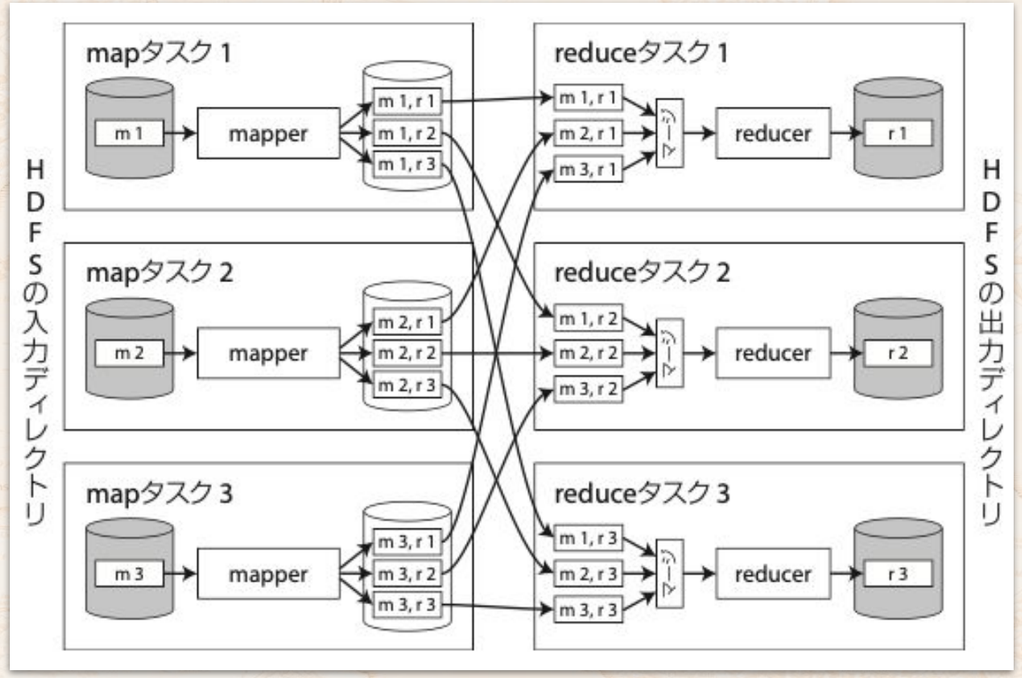
- Unixコマンドでのデータ処理に近い
  - $ cat access.log | grep “xxx.yyy” | awk ‘{ print $2; }’ | sort | uniq
- UNIX哲学
  - 端的な機能を持つ様々なコマンドを用意
  - 個々のコマンドの標準入出力をパイプでつなぐ
  - シェルの裏側ではコマンドごとにプロセスが起動していて、プロセス間のストリーム処理をしている
    - ひと昔前の東大情報科学科のOS演習では、シェルをCで実装する課題があった。プロセス起動、同期(mutex、signal)、I/Oを同時に扱うため、難しい課題
- これを分散（複数ノード）で実行するには？

分散バッチ処理

- Spark (2009)
  - Microsoft DryadLINQ (2008)のアイデアがベース。
  - UNIXコマンドのような命令（小さなプログラム）を並列・分散実行するイメージ
- MapReduce (Google 2004, Hadoop 2006)
  - Mapper: (key, value)のペアを出力する。Reducer: 同じkeyに属するvalueを集めて集計結果を出力
  - フレームワーク側が自動で分散実行
    - MapSide Join, Broadcast/Hash Joinなど様々なテクニックが誕生。SQLも実行できる(Hive)



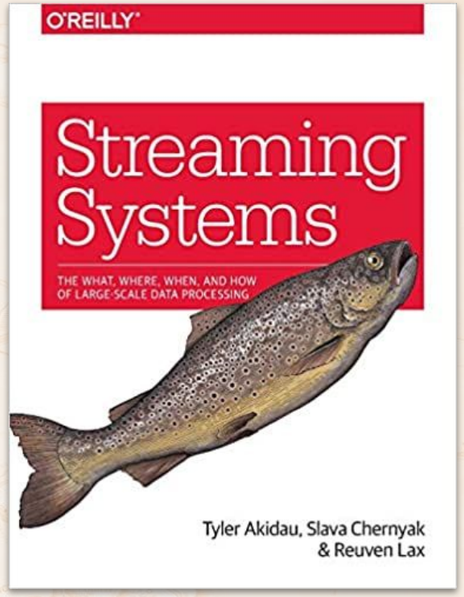
DryadLINQ（2008）

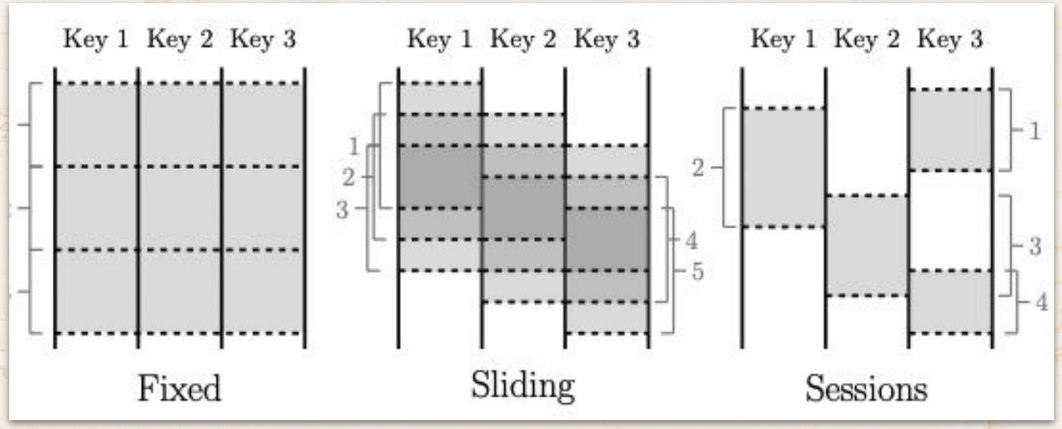


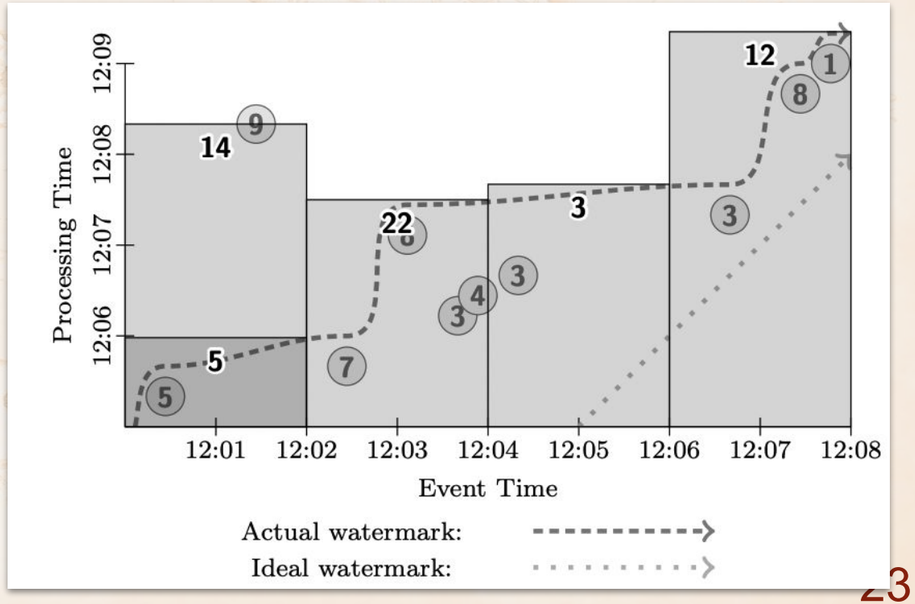
MapReduce

ストリーム処理

- バッチ処理
  - 保存されているデータに対してクエリを実行
- ストリーム処理
  - クエリをデータの入力側に送り、常時クエリを実行し続ける
  - あるいは、データの変更をとらえ処理する
    - マイクロバッチ、CDC (Change Data Capture)
- Dataflow Model (Google, VLDB 2015)
  - 時系列データ処理方法の分類・パターンの定義
  - 遅れてやってくる(late arraival)データの処理を埋め合わせる必要性を提示
    - event time, processing timeをwatermarkで管理
  - Apache Beamなどに実装されている

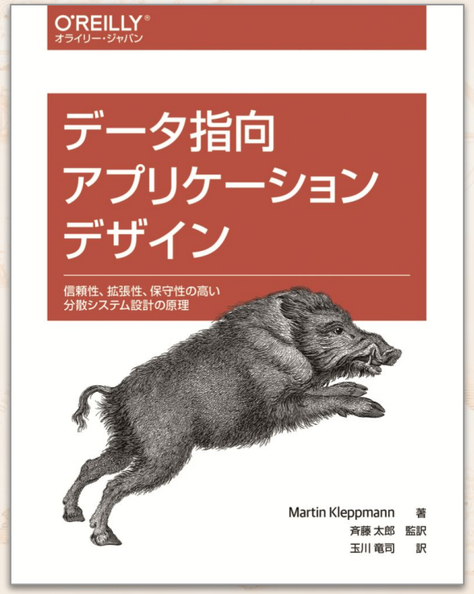




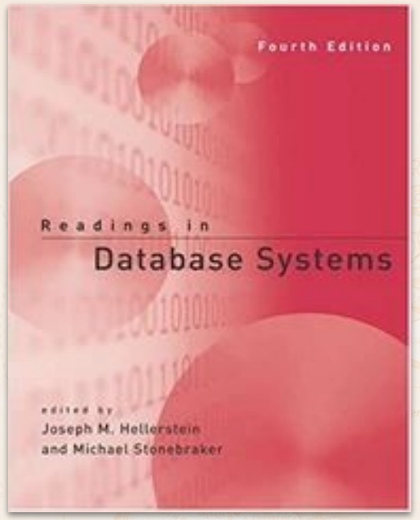


「データの複雑さ」に向き合う

データモデル、クエリ言語の変遷



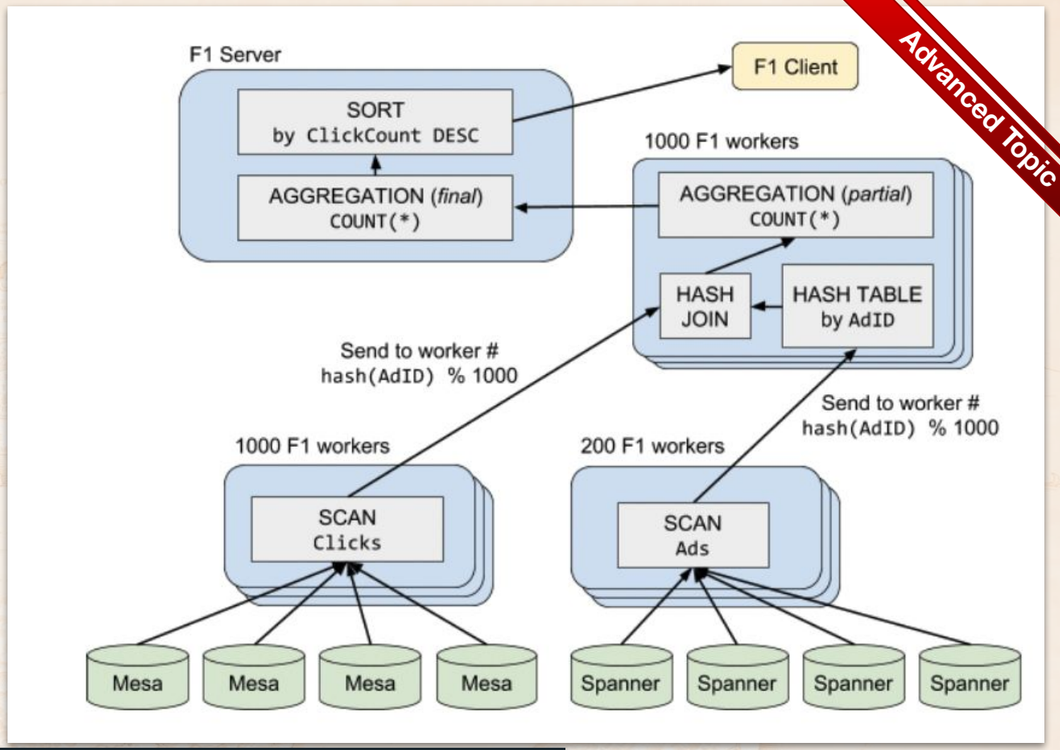
データモデルの多様性とその歴史



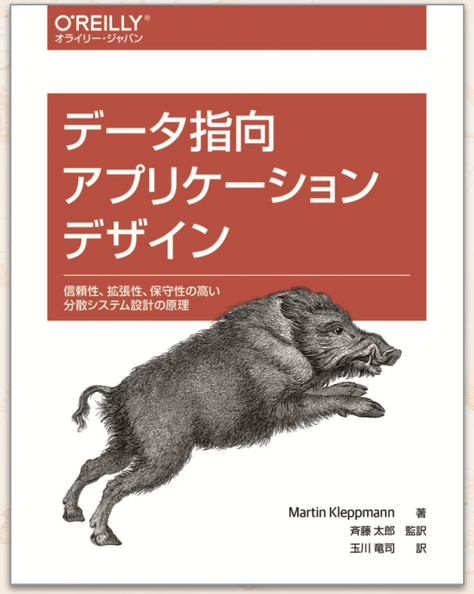
- JSON、XML、テーブル、グラフ（ノードとエッジ）など
- 1970年代から議論が続く （詳しい歴史はRedbookにも）
  - ネットワークモデル (CODASYL)
  - 階層モデル
- リレーショナルモデルによるRDBMSが圧倒
  - 種々のデータ形式(JSON、XMLなど）のサポートが、次々に RDBMSに取り込まれていった
  - その理由：データベースにとっては、DBMSの品質（トランザクションサポートなど）が重要で、新しいモデルが生まれても、ユーザーにとって本格的に使うまでの道のりが長かった。
- インピーダンスミスマッチ（Impedance mismatch）
  - アプリケーションで使いたい表現（構造化データ、オブジェクトなど）と、データベースに格納される形態（テーブル）がマッチしない状況
- NoSQL
  - SQL、リレーショナルモデルへの不満から、ドキュメントモデルなども生まれる
  - バズワードとして広まったが、次第に**Not Only SQL**として解釈されるように
- 歴史から学んだ最近の良い事例
  - GraphQLは**RDBMS、SQLとはむやみに喧嘩しない**アプローチ。RDBMS、SQLは利用しつつも、アプリケーション側が欲しい形でデータを取得するのをサポート

インターフェースとしてのSQLの価値

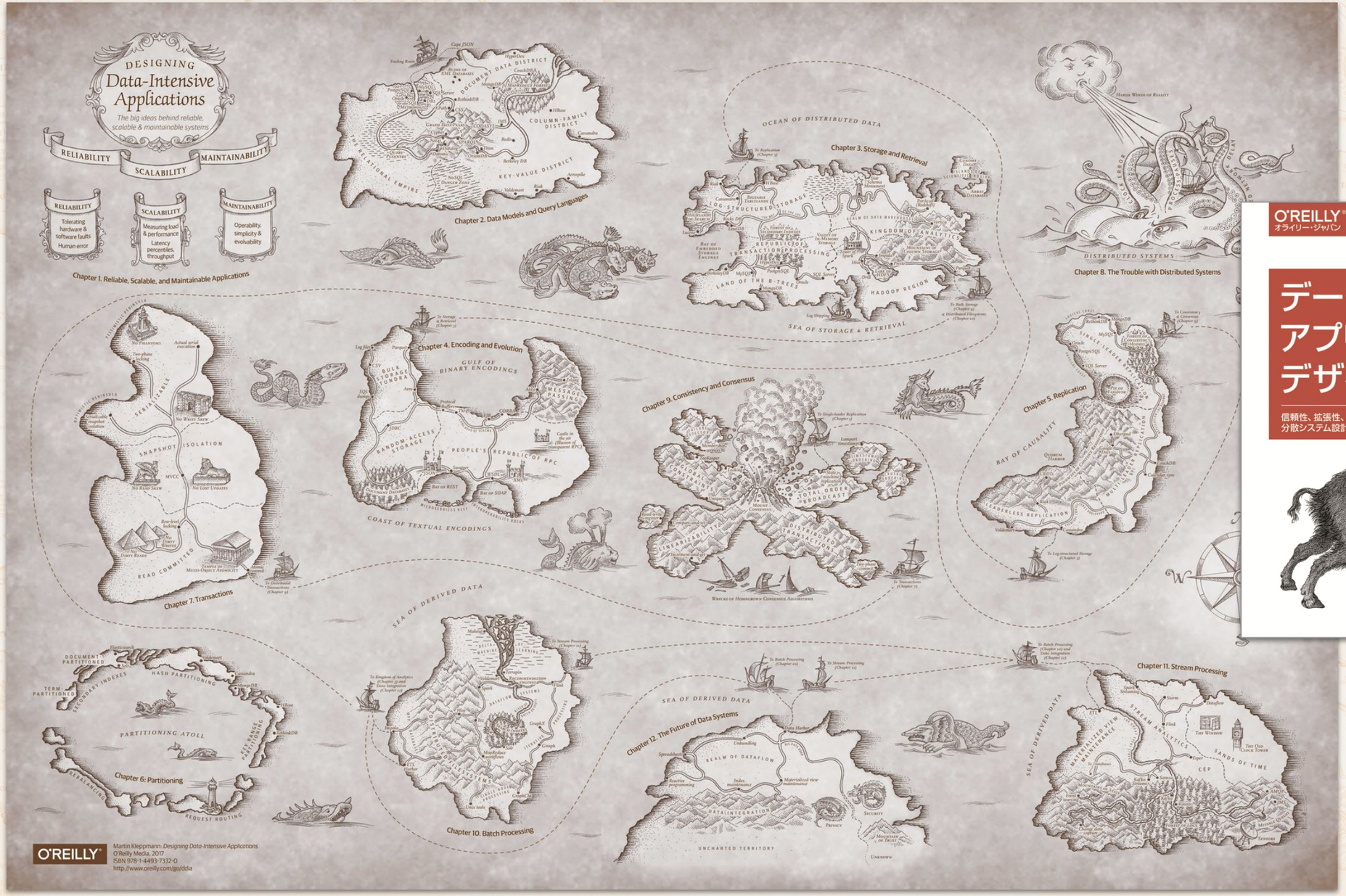
- SQL
  - RDBMS専用のインターフェースだったが。。。
- Google F1（VLDB2018）
  - Google社内の多様なストレージにSQLでアクセス
  - ZetaSQL:共通SQL実装
    - BigQueryにも使われている
- Meta
  - 共通SQL + Velox処理エンジン (CIDR 2023)
- Trino Distributed SQL Engine
  - 種々のデータソースへのコネクタを提供
  - ストレージ実装を持たないSQLエンジン

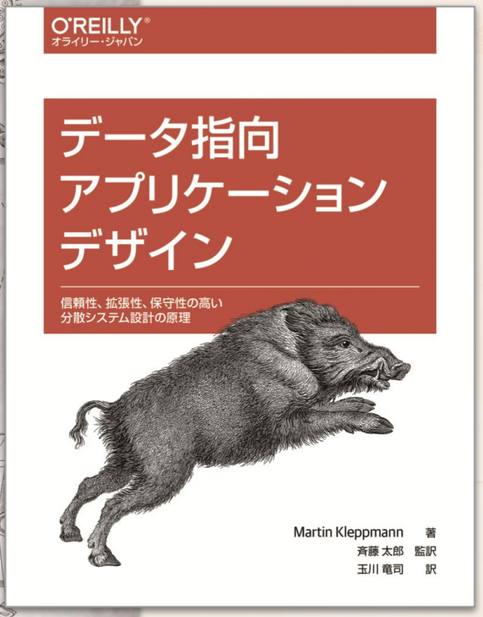


分散データシステムの世界



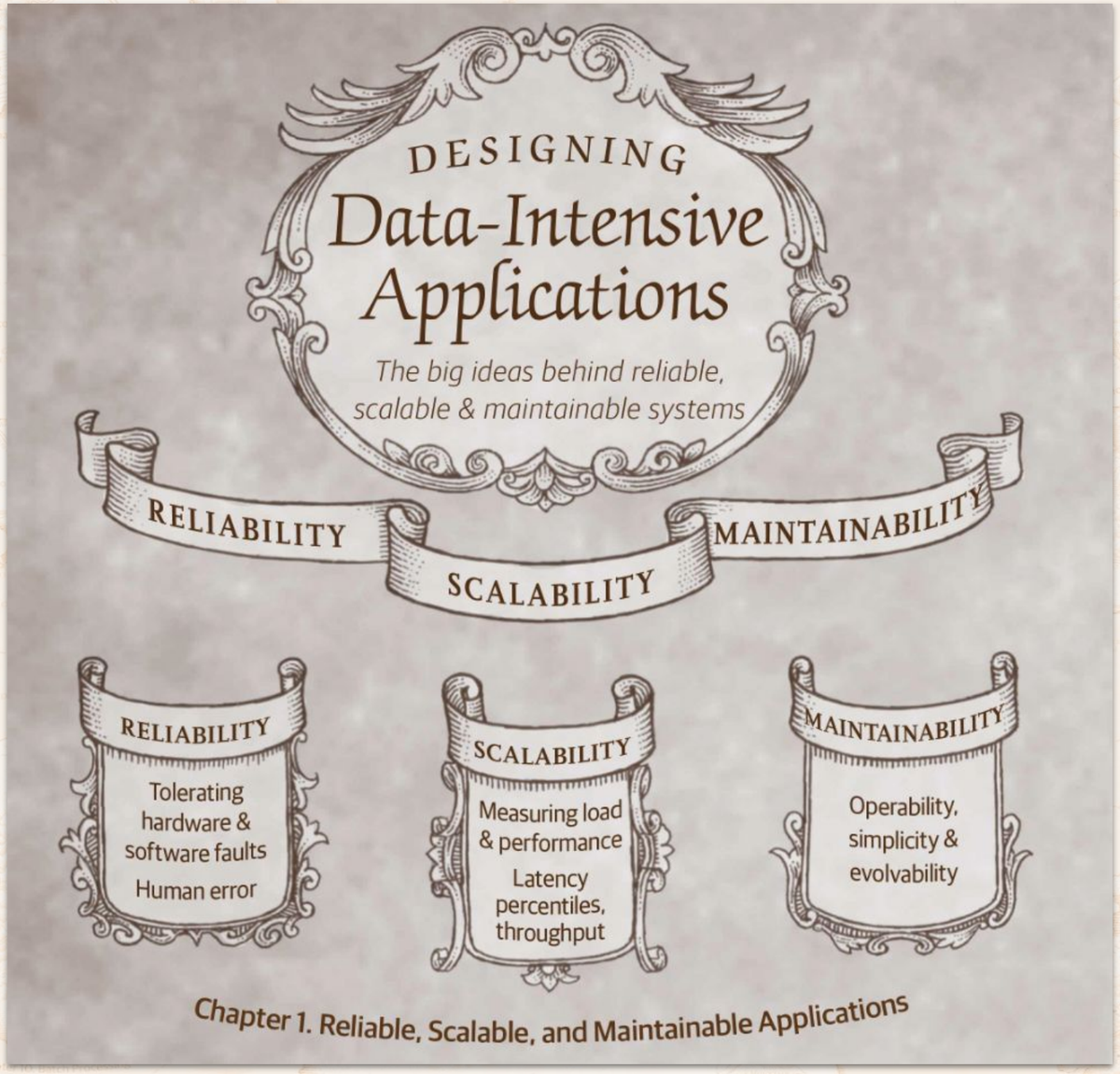
分散データシステムの世界は広大



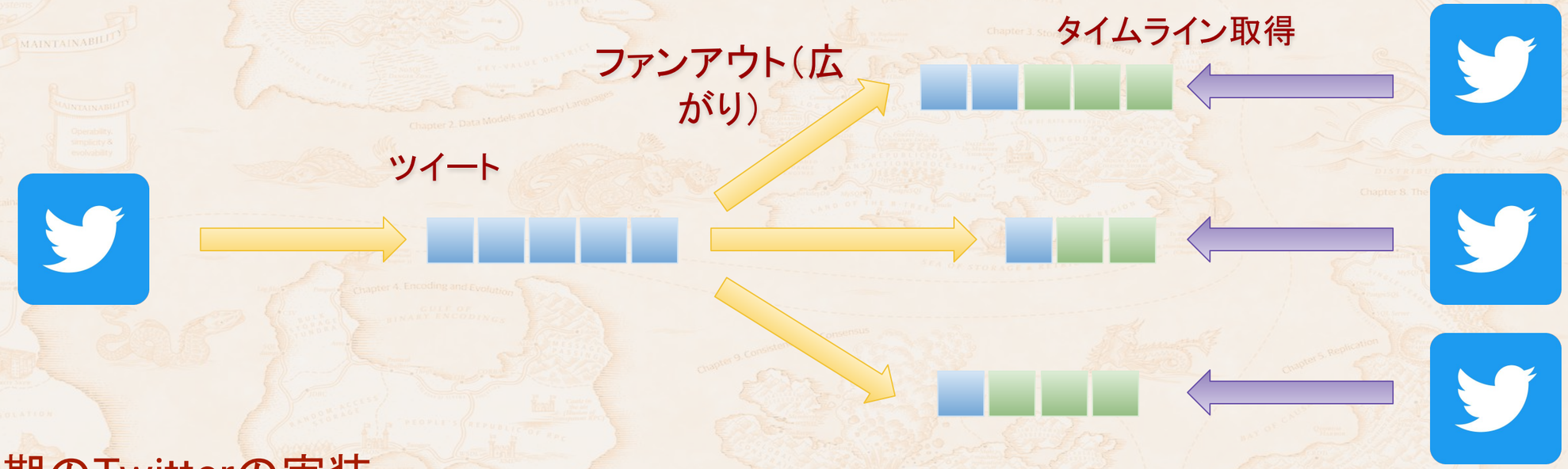


データ指向がわかるとChapter 1の本質も見えてくる

- 信頼性
- スケーラビリティ
- メンテナンス性



具体例:ツイート配信の実装 (Fanout Service)



- 1:初期のTwitterの実装
  - ツイートをグローバルなコレクション(DB)に格納。ユーザーがタイムラインを見るたびにフォロアーのツイートをDBから検索 （**読み込み負荷が高い**）
- 2:(改良)ツイート時にフォロアーのタイムラインキャッシュにもツイートを送信
  - **書き込み負荷は高い**が、タイムラインの**読み込み時の負荷が2桁減った**
- 極端な例: Elon Muskは128Mフォロアー (= 128M ファンアウト)
  - ツイートするたびにフォロアーのタイムラインキャッシュに1.28億回の書き込み？

128M followersの割にElon Muskのインプレッションが低い問題

Advanced Topic

- ファンアウトサービスが高負荷でクラッシュして、95%のツイートが配信されていなかった

- データ指向アプリケーション的に興味深い
- 信頼性
  - 一部のサービスがクラッシュしてもサービスは動き続けている。障害への対応
- メンテナンス性
  - 数千人規模のエンジニアのレイオフをしても耐えられる。運用の自動化、知識の継承、維持ができている（？）
- スケーラビリティ
  - 数億回の読み書きはいずれにしても発生。分散RPC (Finagle) 、レプリケーション、スロットリング（性能調整）はどうなっているか？

SLO: システムのパフォーマンスをどう測るか？

- 平均値を見るだけでは、データ規模が大きく性能が遅くなりやすい重要顧客の様子が見えてこない。
  - 95、99パーセンタイル (p95, p99) などを見てSLOを決める
- SLO (Service Level Objective)
  - Amazonでの例: p99.9でのレスポンスタイムを一定値以下に。100ms 遅くなれば、売り上げが1%下がるというデータがある（2022Q4にonline storeは$64Bの収入。その1% = $640M)
    - ただし、p99.99の極端な領域での改善は、コストの割に利益を十分生まないと判断

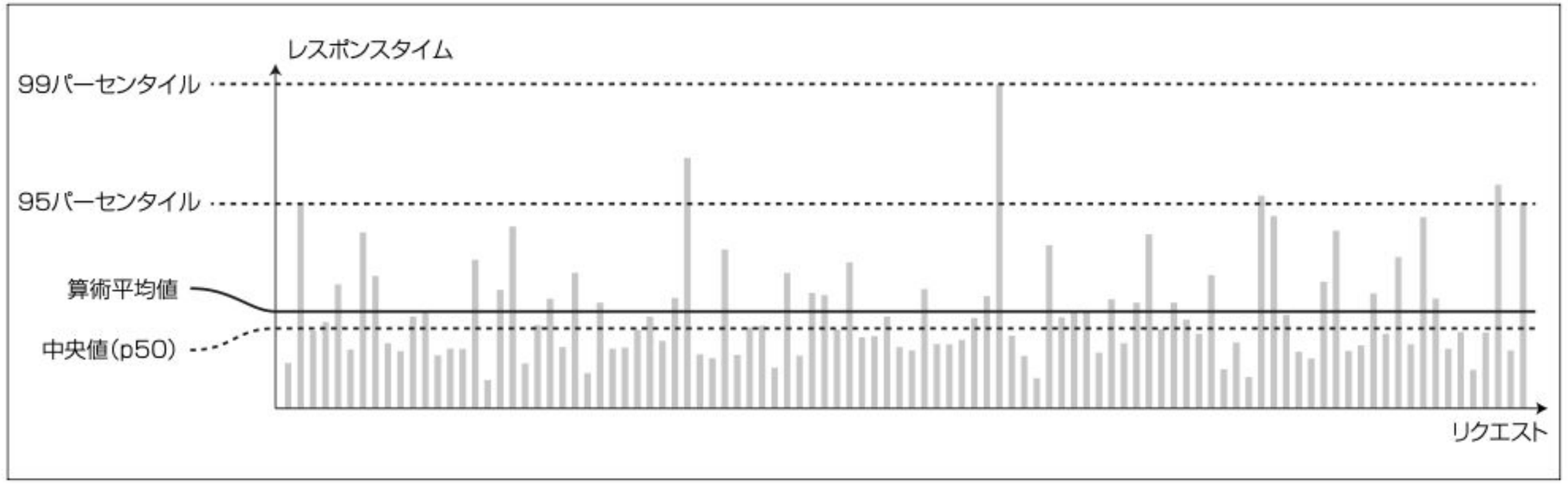
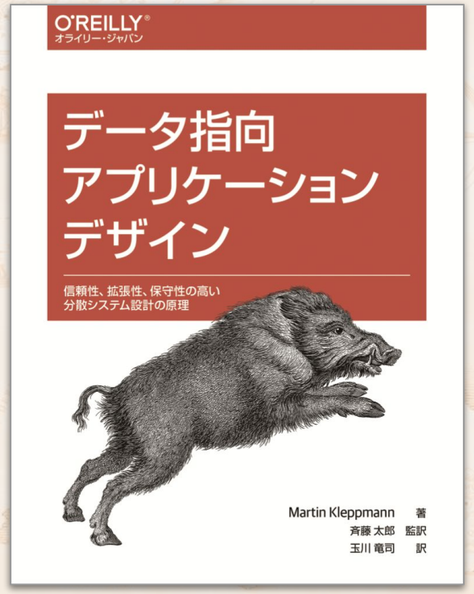
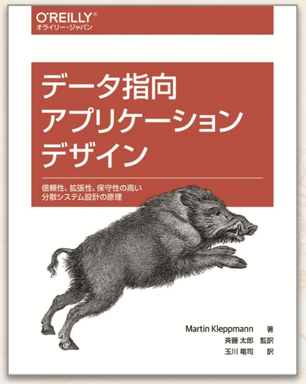


図1-4　平均とパーセンタイルの様子。100回サンプリングされたあるサービスへのリクエストのレスポンスタイム

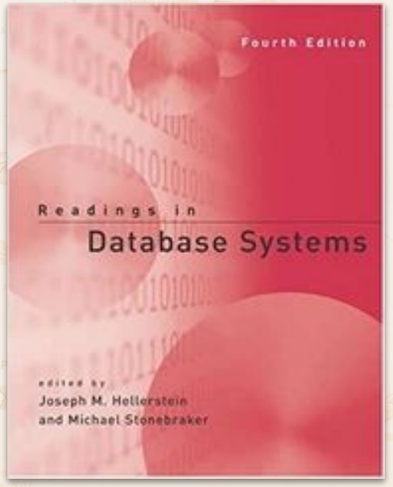
おわりに

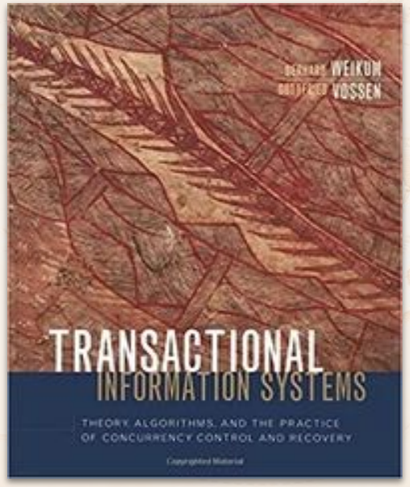

O'REILLY
オライリー・ジャパン
データ指向
アプリケーション
デザイン
信頼性、拡張性、保守性の高い
分散システム設計の原理
Martin Kleppmann 著
斉藤 太郎 監訳
玉川 竜司 訳

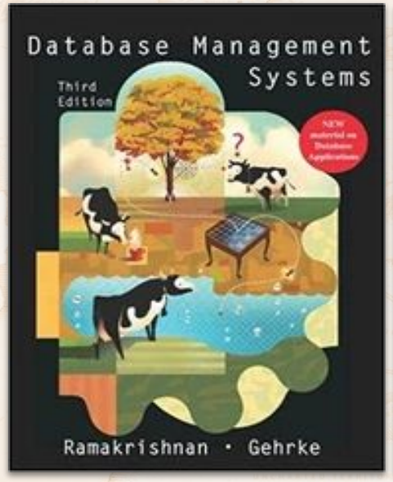
分散データシステム入門の決定版

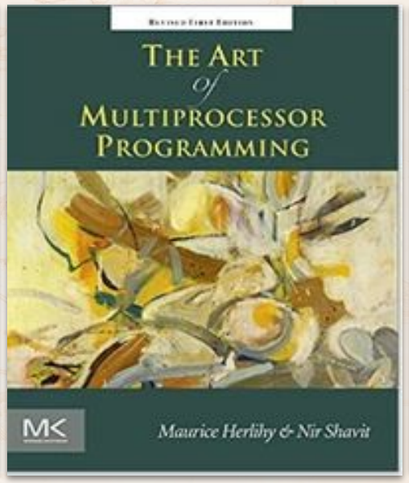


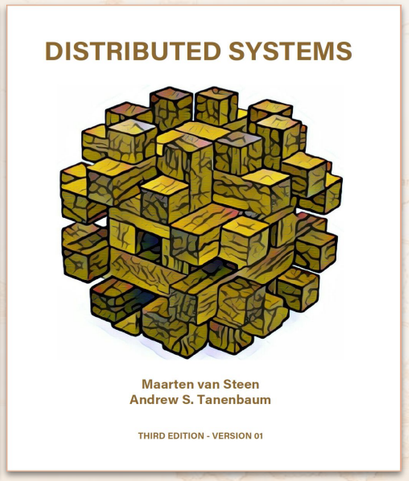
- この分野の教科書は過去10年間存在しなかった (2007-2017)
  - 660ページと重厚だが。。。(鈍器？)
- 関連各分野の教科書はさらに分厚い
  - Readings in Database Systems (Redbook) データベースの歴史、重要論文リスト (878 pages)
  - Transactional Information Systems トランザクション理論 (872 pages)
  - Database Management Systems データベース分野の基礎知識 (1104 pages)
  - The Art of Multiprocessor Programming 並列計算、マルチコアでの同期処理など (576 pages)
  - Distributed Systems 分散システムの基礎 (vector clock, レプリケーションなど) (612 pages)
- **「データ指向アプリケーションデザイン」**
  - **論文、教科書、OSS技術から得られる分散データシステム分野の知見の集大成**









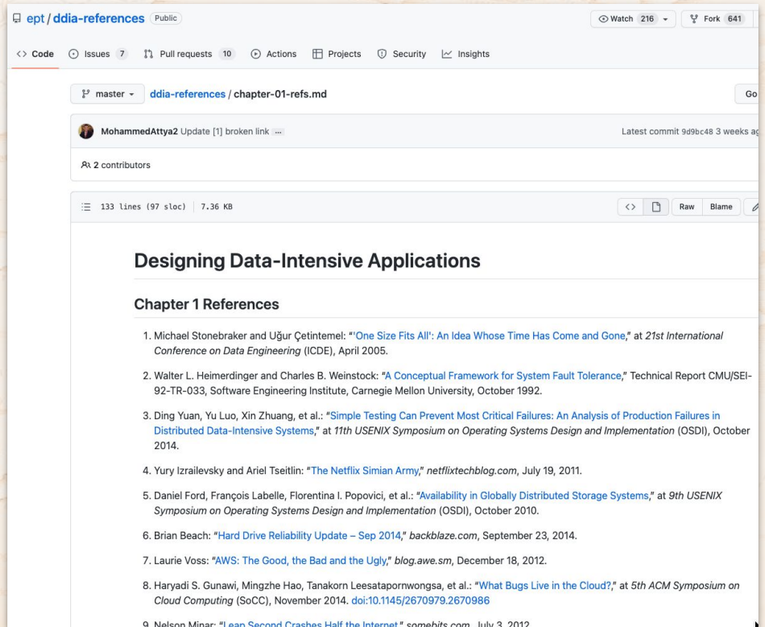


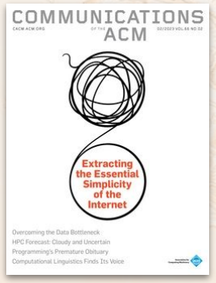
データ指向アプリケーションデザインができるまで

- 2014年から4年間書き続けている
  - 第2版も検討しているらしいが、まだ目処が経っていないとのこと
  - またしばらく、教科書のない空白の10年になりそう。
    - 自分で情報収集する必要あり

まとめ：より深い世界に踏み込むための手引き

- データ分野全般をカバーしたガイドブック
  - 各章ごとに50〜100本程度の引用がある
    - GitHub: ept/ddia-references
- 各分野の基礎知識を抑えよう！
  - 論文からも情報収集できるように
  - Industrial paperが楽しくなります
- Audible (英語版：20時間) もあり
  - 聞き流すと楽





参考

- ストリーム処理、導出データ処理関連の論文リスト
  - https://github.com/xerial/streamdb-readings
- Twitter #dbreadings