# Databases

<table>
<thead>
<tr>
<th>Operational</th>
<th>Scale up</th>
<th>Scale out</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL™</td>
<td></td>
<td>cassandra</td>
</tr>
<tr>
<td>TERADATA</td>
<td></td>
<td>Spark™</td>
</tr>
</tbody>
</table>

© DataStax, All Rights Reserved.
Operational + Scale Out
Stuff everyone agrees on
Stuff (Almost) Everyone Agrees On

1. Eventual Consistency is useful
CAP theorem refresher

- Consistency, Availability, Partition tolerance: CP or AP
- What happens if a node goes down?
  - CP: I want to wait for a new master to get elected before processing requests
  - AP: I’d rather keep handling requests (possibly with stale data) and reconcile asynchronously
Eventual Consistency (CP edition)
Default consistency levels

1. Cassandra: Eventual
2. Dynamo: Eventual
3. CosmosDB: Eventual ("Session")
4. Spanner: ACID (no EC)
Stuff (Almost) Everyone Agrees On

1. Eventual Consistency is useful
2. Automatic partitioning doesn’t work
Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems

Andrew Pavlo  
Brown University  
pavlo@cs.brown.edu

Carlo Curino  
Yahoo! Research  
krl@yahoo-inc.com

Stan Zdonik  
Brown University  
sbz@cs.brown.edu

H-Store (2012)
Partitioning approaches

1. Cassandra: Explicit
2. Dynamo: Explicit
3. CosmosDB: Explicit
4. Spanner: Explicit
Stuff (Almost) Everyone Agrees On

1. Eventual Consistency is useful
2. Automatic partitioning doesn’t work
3. SQL is a pretty okay query language
So, how do I query the database?

It's not a database. It's a key-value store!

Ok, it's not a database. How do I query it?

You write a distributed map reduce function in Erlang!

Did you just tell me to go fuck myself?

I believe I did, Bob.
Query APIs

1. Cassandra: CQL, inspired by SQL
2. DynamoDB: Actually still pretty first-gen NoSQL
3. CosmosDB: “SQL”
4. Spanner: “SQL”
Stuff (Almost) Everyone Agrees On

1. Eventual Consistency is useful
2. Automatic partitioning doesn’t work
3. SQL is a pretty okay query language
4. … that’s about it
Stuff not everyone agrees on

(But I’m right)
Schema is good
Scaling Big Data Mining Infrastructure: The Twitter Experience

Jimmy Lin and Dmitriy Ryaboy
Twitter, Inc.
@lintool @squarecog

ABSTRACT
The analytics platform at Twitter has experienced tremendous growth over the past few years in terms of size, complexity, number of users, and variety of use cases. In this paper, we discuss the evolution of our infrastructure and the development of capabilities for data mining on "big data". One important lesson is that successful big data mining in practice is about much more than what most academics would consider data mining: life "in the trenches" is occupied by much preparatory work that precedes the application of data mining algorithms and followed by substantial effort to turn preliminary models into robust solutions. In this context, we discuss two topics: First, schemas play an important role in helping data scientists understand petabyte-scale data stores, but they're insufficient to provide an overall "big picture" of the data available to generate insights. Second, we observe that a major challenge in building data analytics platforms stems from the heterogeneity of the various components that must be integrated together into production workflows—we refer to this as "plumbing". This paper has two goals: For practitioners, we hope to share our experiences to flatten bumps in the road for those who come after us. For academic researchers, we hope to provide a broader context for data mining in production environments, pointing out opportunities for future work.

A little about our backgrounds: The first author is an Associate Professor at the University of Maryland who spent an extended sabbatical from 2010 to 2012 at Twitter, primarily working on relevance algorithms and analytics infrastructure. The second author joined Twitter in early 2010 and was first a tech lead, then the engineering manager of the analytics infrastructure team. Together, we hope to provide a blend of the academic and industrial perspectives—a bit of ivory tower musings mixed with "in the trenches" practical advice. Although this paper describes the path we have taken at Twitter and is only one case study, we believe our recommendations align with industry consensus on how to approach a particular set of big data challenges.

The biggest lesson we wish to share with the community is that successful big data mining is about much more than what most academics would consider data mining. A significant amount of tooling and infrastructure is required to operationalize vague strategic directives into concrete, solvable problems with clearly-defined metrics of success. A data scientist spends a significant amount of effort performing exploratory data analysis to even figure out "what's there"; this includes data cleaning and data munging not directly related to the problem at hand. The data infrastructure engineers work to make sure that productionized workflows operate smoothly, efficiently, and robustly, reporting errors and alerting responsible parties as necessary. The "core" of what
Schema confusion

{"userid": "2452347",
 "name": "jbellis",
 ...
 }

{"userid": 2452348,
 "name": "jshook",
 ...
 }

{"user_id": 2452349,
 "name": "jlacefield",
 ...
 }
Schema support

- Cassandra: schema-mandatory
- DynamoDB: schema-optional
- CosmosDB: schema-free
- Spanner: schema-mandatory
A detailed look
Thomas Sowell

There are no solutions. Only tradeoffs.
Cassandra
CREATE TABLE notifications (  
target_user text,  
notification_id timeuuid,  
source_id uuid,  
source_type text,  
activity text,  
PRIMARY KEY (target_user, notification_id)  
)  
WITH CLUSTERING ORDER BY (notification_id DESC);
<table>
<thead>
<tr>
<th>target_user</th>
<th>notification_id</th>
<th>source_id</th>
<th>source_type</th>
<th>activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>nick</td>
<td>e1bd2bcb-</td>
<td>d972b679-</td>
<td>photo</td>
<td>tom liked</td>
</tr>
<tr>
<td>nick</td>
<td>321998c-</td>
<td>d972b679-</td>
<td>photo</td>
<td>jake commented</td>
</tr>
<tr>
<td>nick</td>
<td>ea1c5d35-</td>
<td>88a049d5-</td>
<td>user</td>
<td>mike created account</td>
</tr>
<tr>
<td>nick</td>
<td>5321998c-</td>
<td>64613f27-</td>
<td>photo</td>
<td>tom commented</td>
</tr>
<tr>
<td>nick</td>
<td>07581439-</td>
<td>076eab7e-</td>
<td>user</td>
<td>tyler created account</td>
</tr>
<tr>
<td>mike</td>
<td>1c34467a-</td>
<td>f04e309f-</td>
<td>user</td>
<td>tom created account</td>
</tr>
</tbody>
</table>
Collections

CREATE TABLE users (  
id uuid PRIMARY KEY,  
name text,  
state text,  
birth_date int,  
email_addresses set<text>  
);
User-defined Types

CREATE TYPE address (  
    street text,  
    city text,  
    zip_code int,  
    phones set<text>  
)

CREATE TABLE users (  
    id uuid PRIMARY KEY,  
    name text,  
    addresses map<text, address>  
)

SELECT id, name, addresses.city, addresses.phones FROM users;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>addresses.city</th>
<th>addresses.phones</th>
</tr>
</thead>
<tbody>
<tr>
<td>63bf691f</td>
<td>jbellis</td>
<td>Austin</td>
<td>{'512-4567', '512-9999'}</td>
</tr>
</tbody>
</table>
JSON

```
INSERT INTO users JSON
'{"id": "0514e410-",
 "name": "jbellis",
 "addresses": {"home": {"street": "9920 Cassandra Ave",
    "city": "Austin",
    "zip_code": 78700,
    "phones": ["1238614789"]}}}'
```
Without nesting

SELECT * FROM users
WHERE user_id = ?

SELECT * FROM addresses
WHERE user_id = ?

SELECT * FROM phones
WHERE address_id = ?
With nesting

SELECT * FROM users
WHERE user_id = ?
Consistency levels

- LOCAL_ONE, LOCAL_QUORUM, LOCAL_SERIAL
- ONE, QUORUM, SERIAL
- TWO, THREE
- ALL, ANY, EACH_QUORUM
Multi-region

1. (Optionally) synchronous writes locally; async globally
2. Serve reads and writes for any row in any region
3. DR is “free”
4. Client-level support
Notable features

- Lightweight Transactions (Paxos, expensive)
- Materialized Views
- User-defined functions
DynamoDB
CP single partition

- Original Dynamo was AP
- DynamoDB offers Strong/Eventual read consistency
- But

  Conditional writes are the same price as regular writes

  And: “All write requests are applied in the order in which they were received”
Data model

- “Map of maps”
- Primary key, sort key
Data Model

Table: Music

Attributes (name-value pairs)
Multi-region

- New feature (late 2017): Global tables
- Shards and replicates a table across regions
- Each shard can only be written to by its master region
Notable features

- Global indexes
- Change feed
- "DynamoDB Transaction Library"

“A put that does not contend with any other simultaneous puts can be expected to perform \(7N + 4\) writes as the original operation, where \(N\) is the number of requests in the transaction.”
Sidebar: cross-partition txns in AP?

Scalable Atomic Visibility with RAMP Transactions

Peter Bailis, Alan Fekete*, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and *University of Sydney
CosmosDB
Data model:
CP Single Partition

- APIs for the following data models are supported with SDKs available in multiple languages:
  - **SQL API**: A schema-less JSON database engine with rich SQL querying capabilities.
  - **MongoDB API**: A massively scalable *MongoDB-as-a-Service* powered by Azure Cosmos DB platform. Compatible with existing MongoDB libraries, drivers, tools, and applications.
  - **Cassandra API**: A globally distributed Cassandra-as-a-Service powered by Azure Cosmos DB platform. Compatible with existing *Apache Cassandra* libraries, drivers, tools, and applications.
  - **Graph (Gremlin) API**: A fully managed, horizontally scalable graph database service that makes it easy to build and run applications that work with highly connected datasets supporting Open Graph APIs (based on the *Apache TinkerPop specification*, Apache Gremlin).
  - **Table API**: A key-value database service built to provide premium capabilities (for example, automatic indexing, guaranteed low latency, global distribution) to existing Azure Table storage applications without making any app changes.
  - Additional data models coming soon!
“Multi model”

- NOT just “APIs”
- More like MyRocks/MongoRocks than C* CQL/JSON
- What they have in common:
  - Hash partitioning
  - Undefined sorting within partitions; use ORDER BY
Data model

The semantics for partition keys are slightly different to match the semantics of each API, as shown in the following table:

<table>
<thead>
<tr>
<th>API</th>
<th>Partition key</th>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azure Cosmos DB</td>
<td>Custom partition key path</td>
<td>Fixed id</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Custom shared key</td>
<td>Fixed _id</td>
</tr>
<tr>
<td>Graph</td>
<td>Custom partition key property</td>
<td>Fixed id</td>
</tr>
<tr>
<td>Table</td>
<td>Fixed PartitionKey</td>
<td>Fixed RowKey</td>
</tr>
</tbody>
</table>
“Multi model”

● Not all features supported everywhere
  ○ Azure Functions
  ○ Change Feed

● TLDR use SQL/Document API
Query

```
SELECT *
FROM Families f
WHERE f.id = "AndersenFamily"
```

Results

```
[{
    "id": "AndersenFamily",
    "lastName": "Andersen",
    "parents": [
        { "firstName": "Thomas" },
        { "firstName": "Mary Kay" }
    ],
    "children": [
        {
            "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,
            "pets": [{ "givenName": "Fluffy" }]
        }
    ],
    "address": { "state": "WA", "county": "King", "city": "seattle" },
    "creationDate": 1431629472,
    "isRegistered": true
}]
```
SQL support and extensions

```
SELECT c.givenName
FROM Families f
JOIN c IN f.children
WHERE f.id = 'WakefieldFamily'
ORDER BY f.address.city ASC
```

“The language lets you refer to nodes of the tree at any arbitrary depth, like Node1.Node2.Node3.....NodeM”
Query

```sql
SELECT { "state": f.address.state, "city": f.address.city, "name": f.id } FROM Families f WHERE f.id = "AndersenFamily"
```

Results

```json
[
{
   "$1": {
       "state": "WA",
       "city": "seattle",
       "name": "AndersenFamily"
   }
}
]
```

Let's look at the role of $1 here. The `SELECT` clause needs to create a JSON object and since no key is provided, we use implicit argument variable names starting with $1. For example, this query returns two implicit argument variables, labeled $1 and $2.
Consistency Levels

- Strong
- Bounded-staleness
- Session
- Consistent Prefix
- Eventual

Lower latency, higher availability, better read scalability
Consistency Levels

• This is probably still too many ("About 73% of Azure Cosmos DB tenants use session consistency and 20% prefer bounded staleness.")
PNUTS: Yahoo!’s Hosted Data Serving Platform

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerneni
Yahoo! Research
Multi-region

● Local read/writes with async replication between regions
● What about ACID consistency support?
  ○ Fine print: “Azure Cosmos DB accounts that are configured to use strong consistency cannot associate more than one Azure region with their Azure Cosmos DB account”
Notable features

- Everything is indexed
- Stored procedures
  - Including (single-partition) transactions
  - Only for SQL API
- Change feed
Spanner
Data model: CP multi-partition

- “Reuse existing SQL skills to query data in Cloud Spanner using familiar, industry-standard ANSI 2011 SQL.”
  (Actually a fairly small subset)
  (And only for SELECT)
Interleaved/child tables

CREATE TABLE Singers (  
    SingerId   INT64 NOT NULL,
    FirstName  STRING(1024),
    LastName   STRING(1024),
    SingerInfo BYTES(MAX),
) PRIMARY KEY (SingerId);

CREATE TABLE Albums (  
    SingerId     INT64 NOT NULL,
    AlbumId      INT64 NOT NULL,
    AlbumTitle   STRING(MAX),
) PRIMARY KEY (SingerId, AlbumId),  
    INTERLEAVE IN PARENT Singers ON DELETE CASCADE;
JOIN types

Syntax

```
join:
    from_item [ join_type ] [ join_method ] JOIN [ join_hint_expr ] from_item
    [ ON bool_expression ] USING ( join_column [, ...] ) ]

join_type:
    { INNER | CROSS | FULL [OUTER] | LEFT [OUTER] | RIGHT [OUTER] }

join_method:
    { HASH }

join_hint_expr:
    '{@join_hint_key = join_hint_value [, ...] '}

join_hint_key:
    { FORCE_JOIN_ORDER | JOIN_TYPE }
```
Cloud Spanner has three types of replicas: **read-write replicas**, **read-only replicas**, and **witness replicas**. Single-region instances use only read-write replicas, while multi-region instance configurations use a combination of all three types. (To understand why, see: Why read-only and witness replicas?)

The following table summarizes the types of Cloud Spanner replicas and their properties, and the sections below describe each type in more detail.

<table>
<thead>
<tr>
<th>Replica type</th>
<th>Can vote</th>
<th>Can become leader</th>
<th>Can serve reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-write</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Read-only</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Witness</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Multi-region, TLDR

- You can replicate to multiple regions
- Only one region can accept writes at a time
Notable features

● Full multi-partition ACID 2PC (using Paxos replication groups)
The price of ACID (late 2017)

YCSB Throughput (ops/sec, each op 1KB)

- Load Throughput: 114845 (kudu)
- Zipfian Random Read (100 threads): 12663 (kudu), 8612 (spanner)
Writes slow down (indexed) reads

Quizlet:

“Bulk writes severely impact the performance of queries using the secondary index [because] a write with a secondary index updates many splits, which [since Spanner uses pessimistic locking] creates contention for reads that use that secondary index.”
Practical considerations
Cassandra

- Run anywhere you like--but you have to run it
  - But: DataStax Managed Cloud, Instaclustr
  - Also: DataStax Remote DBA
- Storage closely tied to compute
  - But: everyone struggles with this
Multi-cloud

- JP Morgan: “We have seen increasingly all the customers we talk to, almost exclusively large mid-market to large enterprise, all now are embracing multi-cloud as a specific strategy.”
- **Want to avoid cloud lock-in? It's about the database**

As companies barrel into the public cloud, they may want to consider keeping their database separate to preserve freedom from vendor lock-in.
DynamoDB

- Request capacity tied to “partitions” [pp]
  - pp count = max (rc / 3000, wc / 1000, st / 10 GB)

- Subtle implication: capacity / pp decreases as storage volume increases
  - Non-uniform: pp request capacity halved when shard splits

- Subtle implication 2: bulk loads will wreck your planning
“Best practices for tables”

- Bulk load 20M items = 20 GB
- Target 30 minutes = 11,000 write capacity = 11 pps
- Post bulk load steady state 200 req/s = 18 req/pp
- No way to reduce partition count
Ravelin, 2017

You construct a table which uses a customer ID as partition key. You know your customer ID’s are unique and should be uniformly distributed across nodes. Your business has millions of customers and no single customer can do so many actions so quickly that the individual could create a hot key. Under this key you are storing around 2KB of data.

This sounds reasonable.

This will not work at scale in DynamoDb.
Further reading

- **You Probably Shouldn’t Use DynamoDB**
  - Hacker News Discussion

- **The Million Dollar Engineering Challenge**
  - Hacker News discussion
CosmosDB

- Like DynamoDB, but underdocumented
- **Partition and scale in Azure Cosmos DB**
  - Unspecified: max pp storage size, max pp request capacity
Spanner

- DFS architecture means it doesn’t have the provisioning problem
- Priced per node (~$8500 min per 2 TB data, per year) + $3600 / TB / yr
  - Doesn’t appear to be part of the GCP free usage tier
- Competing with Cloud Datastore, Cloud BigTable
Recommendations
My Bias: DataStax Enterprise
Never recommended

- DynamoDB: Data model, feature set is too limiting
- Spanner: All ACID, all the time is too expensive
CosmosDB vs Cassandra

- Rough feature parity
  - Partitioned rows (documents) with nesting
  - Default EC with opt-in to stronger options

- Cassandra
  - Materialized views
  - True multi-model
  - Predictable provisioning

- CosmosDB
  - Stored procedures
  - ORDER BY
  - Change feed
Distributed Systems Reading (We’re hiring)

- Bigtable: A Distributed Storage System for Structured Data
- Dynamo: Amazon’s Highly Available Key-value Store
- Cassandra - A Decentralized Structured Storage System [annotated by Jonathan Ellis]
- Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems
- Calvin: Fast Distributed Transactions for Partitioned Database Systems
- Spanner: Google's Globally-Distributed Database