# NoSQL

COSC430—Advanced Databases

David Eyers

# Learning Objectives

- Understand how to define ACID, BASE, and CAP
- Describe difficulties relational databases can face
- Explain the "database architecture revolution"
- Give advantages and disadvantages of, and uses for:
  - Key-value databases
  - Document databases

# Reminder!

- Read the "Spanner" paper; summarise its key points
  - We will discuss the paper in next week's lecture
  - http://www.cs.otago.ac.nz/cosc430/assignments/spanner.pdf

# Relational databases

• • •

# The "ACID" properties for DB transactions

- **Atomicity**—"all or nothing" if one part of a transaction fails, then the whole transaction fails
- **Consistency**—the database is kept in a consistent state before and after transaction execution
- **Isolation**—one transaction should not see the effects of other, in progress, transactions
- **Durability**—ensures transactions, once committed, are persistent

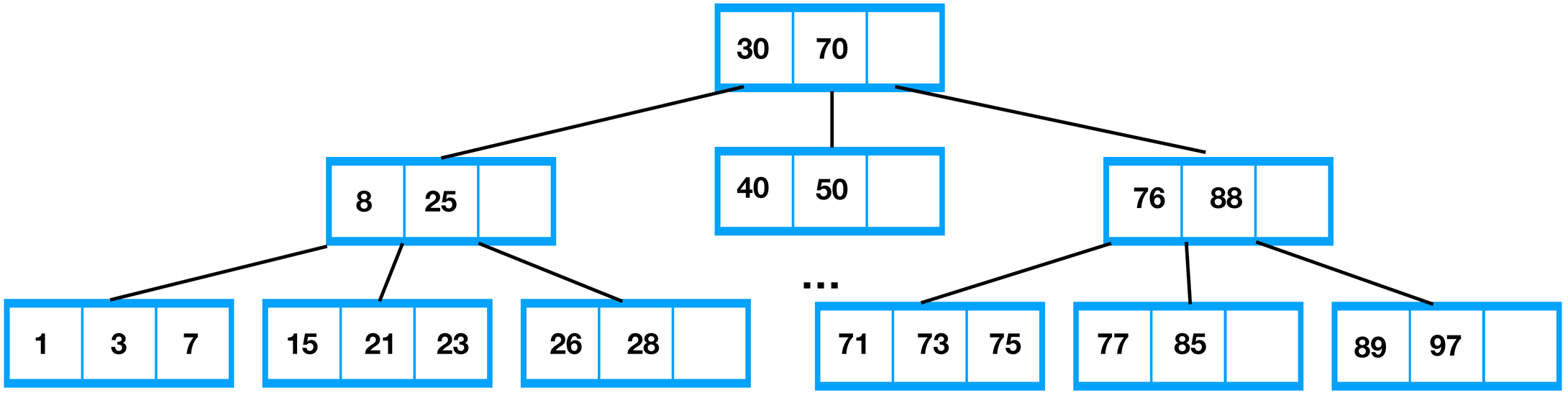
# Typical RDBMs implementations

- Technologies commonly used for speed, yet ACID:
  - disk-oriented storage and indexing structures (e.g., B-Trees)
  - multithreading to hide latency
  - locking-based concurrency control (e.g., two-phase locking)
    - optimistic concurrency control can also be used
  - log based recovery methods (e.g., write-ahead logs)

- More detail in following slides…

# B-Trees allow efficient data management

- Self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions and deletions in $O(\log n)$ time



# Multithreading

- The program can split itself into multiple concurrently executing tasks. **Threads** share resources and are fairly lightweight (compared to operating system **processes**)

- If a single thread has a lot of cache misses then other threads can utilise the idle resources of the machine

- However threads can compete for resources and lead to a decrease in performance…

# Concurrency

- Execution of two (or more) tasks such that they appear to be executing at the same time
  - Think of multitasking applications on a single-core CPU

- Concurrency is not necessarily parallelism—where the tasks are actually executing at the same time
  - Think of multi-core CPUs actually running multiple tasks at once

# Race conditions are a concurrency risk

- Trying to read or modify a variable at the same time

| *T1* | *T2* | *Value* |
|---|---|---|
| | | 0 |
| read | | 0 |
| increment | | 0 |
| write | | 1 |
| | read | 1 |
| | increase | 1 |
| | write | 2 |

**Executes as expected**

| *T1* | *T2* | *Value* |
|---|---|---|
| | | 0 |
| read | | 0 |
| | read | 0 |
| increment | | 0 |
| | increase | 0 |
| write | | 1 |
| | write | 1 |

**Changing order of operations leads to incorrect result**

# Avoiding race conditions using locking

| *T1* | *T2* | *Value* |
|---|---|---|
| | | 0 |
| write_lock | | 0 |
| read | | 0 |
| | write_lock | 0 |
| increment | | 0 |
| | write_lock | 0 |
| write | | 1 |

| *...* | |
|---|---|
| write_lock | 1 |
| unlock | 1 |
| write_lock | 1 |
| read | 1 |
| increment | 1 |
| write_lock | 2 |
| unlock | 2 |

**If the variable is already locked, try to acquire the lock again.**

**This can result in deadlocks**

# Two-Phase Locking (TPL)

- **Expand** phase: acquire all required locks
- **Shrink** phase: release all acquired locks

- Usually three types of locks:
  - shared (read); exclusive (read and write); and range

- Following TPL guarantees **serialisability** of transactions
  - Outcome is the same as if transactions were not executed overlapping in time

# Log-based Recovery

- Log is kept maintaining a sequence of operations
- In the event of failure, the log is replayed

```
<Ti, start>
<Ti, Xj, V1, V2>
<Ti, commit>

...
<Tj, start>
<Tj, Xj, V2, V3>
<Tj, abort>
```

- To recover, the logfile is read from the last entry to the last checkpoint (the last guaranteed consistent state of the system)
- The system makes two lists. The redo list is for all transactions that start and commit. The undo list is for all transactions that start but do not commit or abort
  - Xj—data item; V1—old value; V2—new value

# Architectural revolutions

…

# “The End of an Architectural Era?”

- Traditional RDBMSs evolved from transaction processing systems where ACID properties were the only likely requirement for data handling

- However over the past 35 years to so:
  - Moore’s Law—CPU architectures have changed how they acquire speed
  - New requirements for data processing have emerged
  - Stonebraker et al. (2007), suggest that“one size fits all” DBs end up excelling at nothing; complete DB rethink required

# Emerging application areas

- Text processing (specialised search engines—Google)
- Data warehouses (column stores)
- Stream processing systems
- Scientific and intelligence databases
- Big Data and the Internet of Things
- Smart mobile devices (phones, tablets, watches, etc.)

# ‘Shared’ vs. ‘shared nothing’ approaches

- Main RAM sizes—1970: 1MiB … 2020: 4TiB
- Resource control (the RDBMS is its own OS)
- Grid computing (many low-spec versus big machine)
- High availability (single machine recovered from tape backup versus hot standby)
- Tuning (computer time/memory versus user time costs)

- Many areas are moving towards a distributed model...

# CAP Theorem

- In distributed computing, **choose two** of:
  - **Consistency**—every read receives the most recent data
  - **Availability**—every read receives a response
  - **Partition tolerance**—system continues if network goes down

- Situation is actually more subtle than implied above
  - Can adaptively chose appropriate tradeoffs
  - Can understand semantics of data to choose safe operations

# BASE

- Give up consistency and we can instead get:
  - **Basic Availability**—through replication
  - **Soft state**—the state of the system may change over time
    - This is due to the eventual consistency…
  - **Eventual consistency**—the data will be consistent eventually
    - … if we wait long enough
    - (and probably only if data is not being changed frequently)

# ACID versus BASE Example (1/2)

- Suppose we wanted to track people's bank accounts:

```
CREATE TABLE user (uid, name, amt_sold, amt_bought)
CREATE TABLE transaction (tid, seller_id, buyer_id, amount)
```

- ACID transactions might look something like this:

```
BEGIN
   INSERT INTO transaction(tid, seller_id, buyer_id, amount);
   UPDATE user SET amt_sold=amt_sold + amount WHERE id=seller_id;
   UPDATE user SET amt_bought=amt_bought + amount WHERE id=buyer_id;
END
```

# ACID versus BASE Example (2/2)

- If we consider `amt_sold` and `amt_bought` as estimates, transaction can be split:

```
BEGIN
   INSERT INTO transaction(tid, seller_id, buyer_id, amount);
END
BEGIN
   UPDATE user SET amt_sold=amt_sold + amount WHERE id=seller_id;
   UPDATE user SET amt_bought=amt_bought + amount WHERE id=buyer_id;
END
```

- Consistency between tables is no longer guaranteed
- Failure between transactions may leave DB inconsistent
- See: https://queue.acm.org/detail.cfm?id=1394128 (2008)

# Key-value databases

• • •

# Overview of key-value databases

- Unstructured data (i.e., schema-less)
- Primary key is the only storage lookup mechanism
- No aggregates, no filter operations
- Simple operations such as:
  - **Create**—store a new key-value pair
  - **Read**—find a value for a given key
  - **Update**—change the value for a given key
  - **Delete**—remove the key-value pair

# Advantages

- Simple
- Fast
- Flexible (able to store any serialisable data type)
- High scalability
- Can engineer high availability

# Disadvantages

- Stored data is not validated in any way
  - NOT NULL checks
  - colour versus color
- Complex to handle consistency
  - Checking consistency becomes the application's problem
- No relationships—each value independent of all others
- No aggregates (SUM, COUNT, etc.)
- No searching (e.g., SQL SELECT-style) other than via key

# Examples

- Amazon Dynamo (now DynamoDB)
- Oracle NoSQL Database, ... (eventually consistent)
- Berkeley DB, ... (ordered)
- Memcache, Redis, ... (RAM)
- LMDB (used by OpenLDAP, Postfix, InfluxDB)
- LevelDB (solid-state drive or rotating disk)

# "Dynamo: Amazon's Highly Available Key-value Store"

- Just two operations:
  - `put(key, context, object)`
  - `get(key) → context, object`

- Context contains information not visible to caller
  - but is used internally, e.g., for managing versions of the object

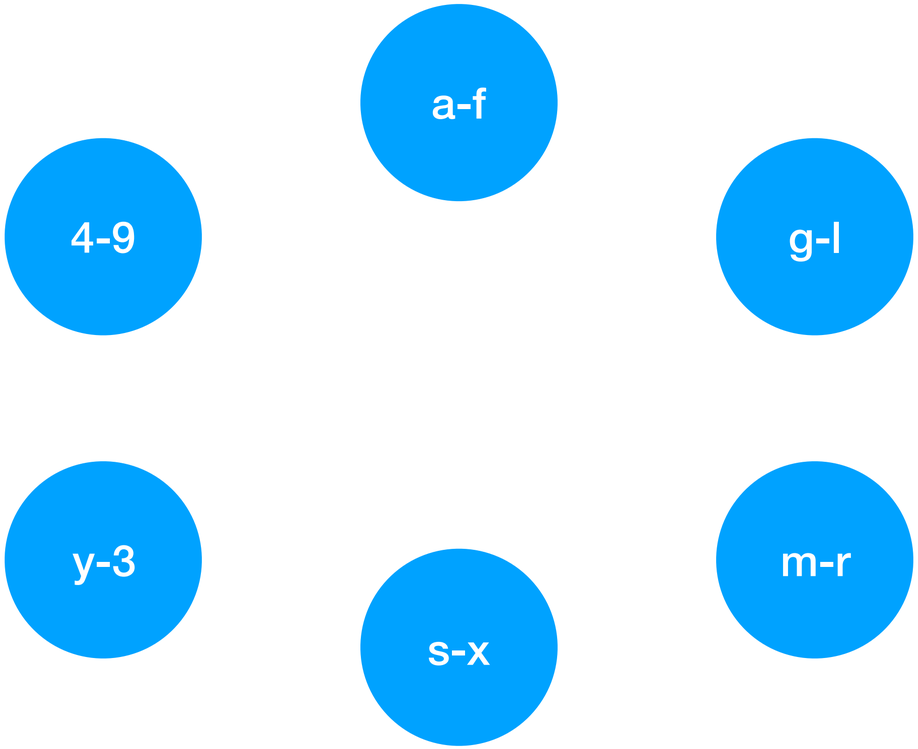
- Objects are typically around 1MiB in size

# Dynamo design

- Reliability is one of the most important requirements
  - Significant financial consequences in its production use
  - Impacts user confidence
- Service Level Agreements (SLAs) are established
- Used within Amazon for:
  - best seller lists; shopping carts; customer preferences; session management; sales rank; product catalogue

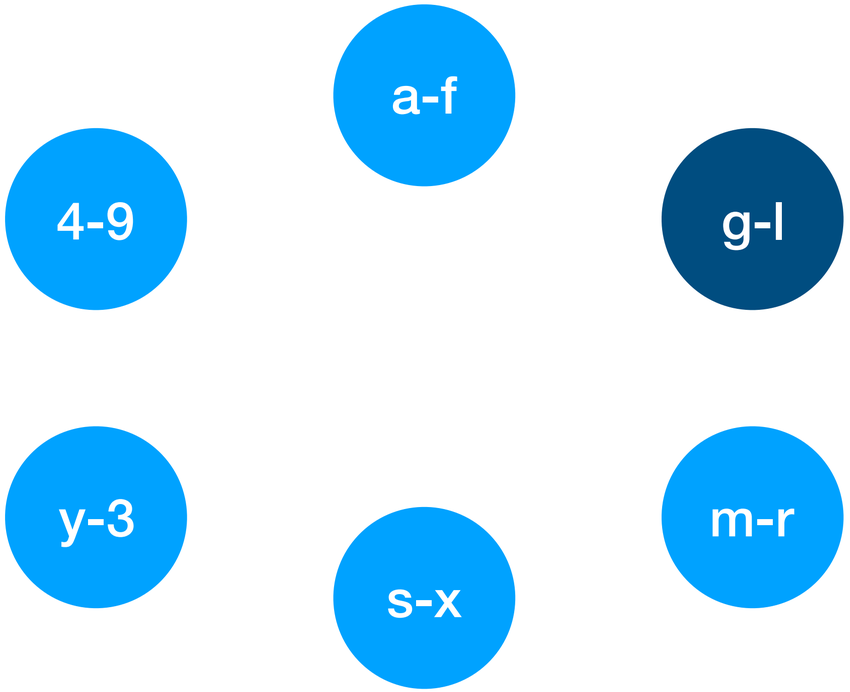
# Dynamo uses partitioning (1/2)

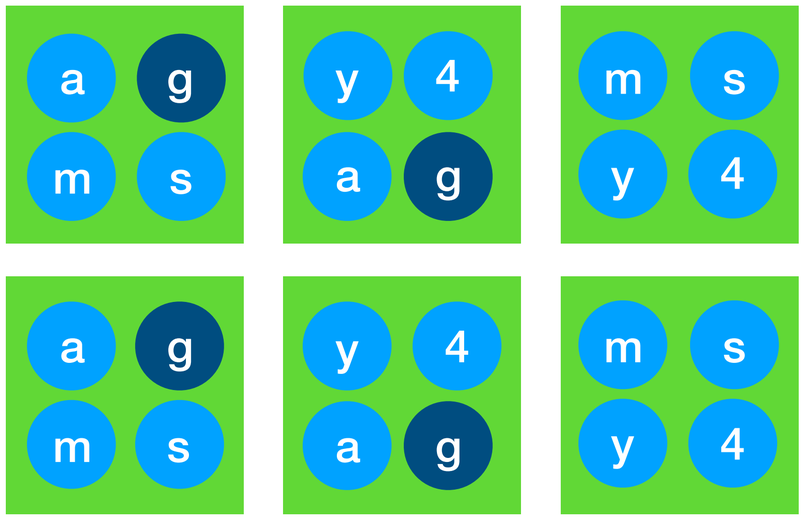
- Consistent hash of the key determines which virtual nodes to store the data within
  - Example here for storing usernames, distributed by first letter



# Dynamo uses partitioning (2/2)

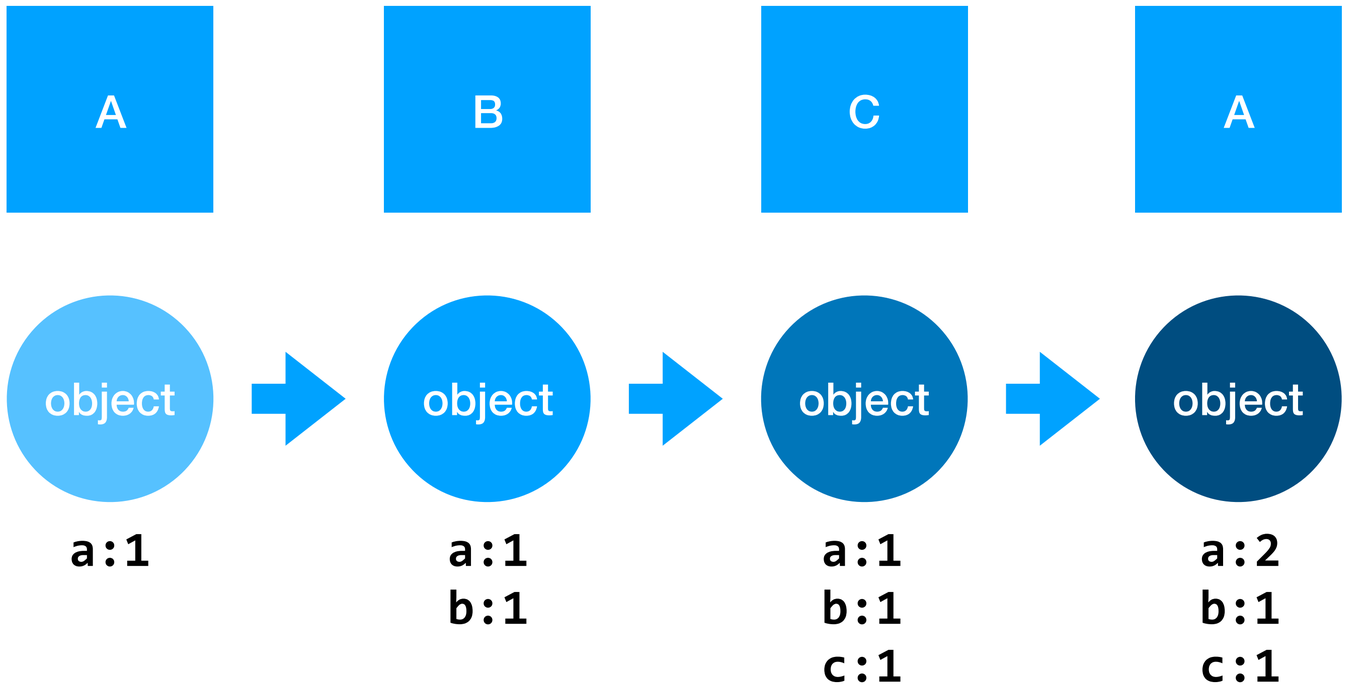
- Each physical node stores a number of virtual nodes
  - Simplified depiction below—consult paper for details
  - N—durability of object (4 here); R/W—#nodes for read/write



# Dynamo manages object versioning

- Metadata stored with objects contains a **vector clock**
  - Each time object is modified the node increments its counter
    - (Shows best-case: updates reach nodes a,b,c before next update)



# Document Databases

• • •

# An example Document in XML format

- XML can be validated against XML schemas (and previously DTDs)

```
<contacts>
    <contact>
        <firstname>David</firstname>
        <lastname>Eyers</lastname>
        <phone type="Cell">+64 21 1234 5678</phone>
        <phone type="Work">+64 3 479 5749</phone>
        <address>
            <type>Work</type>
            <office>1.25</office>
            <street1>133 Union Street East</street1>
            <city>Dunedin</city>
            <state>Otago</state>
            <postcode>9016</postcode>
            <country>NZ</country>
        </address>
    </contact>
    <contact> ... </contact>
</contacts>
```

# An example document in JSON format

- JSON uses native programming types to store data
  - string, number, boolean, list, object, *null*, etc.

```
[
    {
        "firstname":"David",
        "lastname": "Eyers",
        "phone":{
            "cell": "+64 21 1234 5678",
            "work": "+64 3 4795749"
        },
        "address": {
            "work":{
                "office": “1.25",
                "street": "133 Union Street East",
                "city": "Dunedin",
                "state": "Otago",
                "postcode": "9016",
                "country": "NZ"
            }
        }
    },
    {
    ...
    }
]
```

# Overview of document databases

- Semi-structured data model
- Storage of documents:
  - typically JSON or XML
  - could be binary (PDF, DOC, XLS, etc.)
- Additional metadata (providence, security, etc.)
- Builds index from contexts and metadata

# Advantages

- Storage of raw program types (JSON/XML)
- Indexed by content and metadata
- Complex data can be stored easily
- No need for costly schema migrations
  - (Always remember that your DB is likely to need to evolve!)

# Disadvantages

- Same data replicated in each document
- Risk inconsistent or obsolete document structures

# Example implementations

- LinkedIn's Espresso
- ElasticSearch
- CouchDB
- MongoDB
- Solr
- RethinkDB
- Microsoft DocumentDB
- PostgreSQL (when used atypically)

# “On brewing fresh espresso…”—design

- Scale and elasticity
- Consistency
- Integration
- Bulk operations
- Secondary indexing
- Schema evolution
- Cost to serve

# Espresso's data model

- Derived from observed usage patterns at LinkedIn
  - Nested entities were present:
    - group of entities that share a hierarchy
    - e.g. songs in an album for an artist
  - Independent entities:
    - have many-to-many relationships

# Espresso's data hiearchy

```
/Database/Table/Document[/Subresource ...]
```

- **Database**—analogous to RDBMS: contains partitioning scheme and metadata
- **Table**—collection of like-schema-ed documents
- **Document**—logically map entities and are schema-ed (like rows in RDBMS tables)
- **Subresource**—if `Document` is a collection, this choses an item from the collection

# Espresso's data hierarchy—examples

```
/Database/Table/Document[/Subresource ...]
```

- /Music/Artist/The_Beatles
  - Group: “The Beatles”
  - Origin: “Liverpool, UK”
  - Active: “1960–1970”
  - Genres: “rock, pop, psychedlia”
- /Music/Artist/Rolling_Stones
  - Group: “The Rolling Stones”
  - Origin: “London, UK”
  - Active: “1962–”
  - Genres: “rock, blues, rock and roll”

# Espresso—querying

- Fields are annotated in the schema to adjust indexing properties
  - Query:

```
/Music/Song/The_Beatles?query=lyrics:“Lucy in the sky”
```

  - Results:

```
/Music/Song/The_Beatles/Sgt._Pepper/Lucy_in_the_Sky_with_Diamonds
/Music/Song/The_Beatles/Magical_Mystery_Tour/I_am_the_Walrus
```

# Espresso—editing data

- Adjust documents via full or partial paths
- Schema changes are versioned
  - Compatibility checks ensure old documents can be updated
  - Each document stores the version number
- Transactional update support
- Maintains a secondary index to support textual queries

# Summary

- Discussed ACID and its associated challenges
- Describe database architecture revolution
  - BASE and CAP considerations contrasting ACID compliance
- Key-value databases
  - explored Amazon's Dynamo
- Document DBs
  - explored LinkedIn's Espresso

- Please read the *Spanner* paper during this week

# References


- A. Pavlo and M. Aslett, "What's Really New with NewSQL?," *ACM SIGMOD Rec.*, vol. 45, no. 2, pp. 45–55, 2016.
- M. Stonebraker, et. al., "The End of an Architectural Era (It's Time for a Complete Rewrite)," *VLDB*, vol. 12, no. 2, pp. 1150–1160, 2007.
- G. DeCandia, et. al., "Dynamo: Amazon's Highly Available Key-value Store," *Proc. Symp. Oper. Syst. Princ.*, pp. 205–220, 2007.
- A. Auradkar, et. al., "Data infrastructure at LinkedIn," *ICDE*, pp. 1370–1381, 2012.
- L. Qiao, et. al., "On brewing fresh espresso: LinkedIn's distributed data serving platform," *SIGMOD*, pp. 1135, 2013.