To be relational, or not ?
That's not the question!

Sergio Bossa
Alex Snaps

Agenda

- Act 1 – Relational databases:
  a difficult love affair.
- Interlude 1 – Problems in the modern era:
  big data and the CAP theorem.
- Act 2 : Non-relational databases: love is in the air.
- Interlude 2 : Relational or non-relational?
  Not the correct question.
- Act 3 : Building scalable apps.
- The End

About us

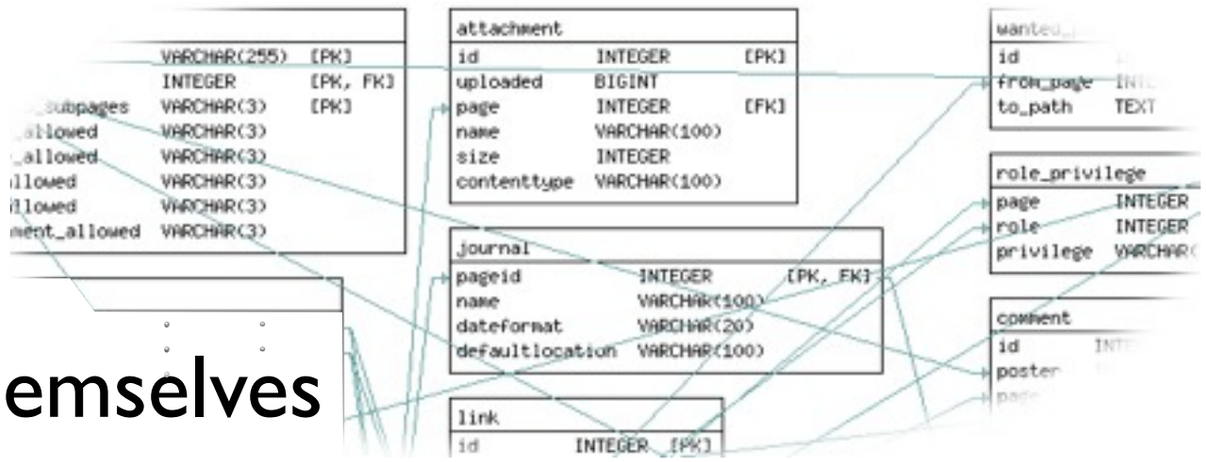
- Sergio Bossa
  - Software Engineer at Bwin Italy.
  - Long time open source contributor.
  - (Micro)Blogger - @sbtourist.
- Alex Snaps
  - Software engineer at Terracotta …
  - … after 10 years of industry experience.
  - www.codespot.net — @alexsnaps

Act I
Relational databases

A difficult love affair …

The relational model

- Defines constraints
  - Finite model
    aka relation variable, relation or table
    - Candidate keys
    - Foreign keys
- Queries are relations themselves
  - Heading & body
- SQL differs slightly from the "pure" relational model



The ACID guarantees

- Let people easily reason about the problem.
  - **Atomic.**
    - We see all changes, or no changes at all.
  - **Consistent.**
    - Changes respect our rules and constraints.
  - **Isolated.**
    - We see all changes as independently happening.
  - **Durable.**
    - We keep the effect of our changes forever.
- Fits a simplified model of our reality.

The SQL ubiquity

- SQL is everywhere
  - Still trying to figure out why my blog uses a relational database to be honest
- SQL is known by everyone
  - Raise your hand if you've never written a SQL query
- ... and if you don't want to
  - ORM are there for you
  - ActiveRecord if you're into Rails
- Simple persistence for all our objects!

Interlude I

Problems of the Modern Era

Big Data

- What's Big Data for you?
- Not a question of quantity.
  - Gigabytes?
  - Terabytes?
  - Petabytes?
- It's all about supporting your business growth.
  - Growth in terms of schema evolution.
  - Growth in terms of data storage.
- Growth in terms of data processing.
- Is your data stack capable of handling such a growth?

CAP Theorem

- **Year 2000**: Formulated by Eric Brewer.
- **Year 2002**: Demonstrated by Lynch and Gilbert.
- **Nowadays**: A religion for many distributed system guys.
- CAP Theorem in a few words:
  - Consistency.
  - Availability.
  - Partition-Tolerance.
  - Pick (at most) two.
- More later.

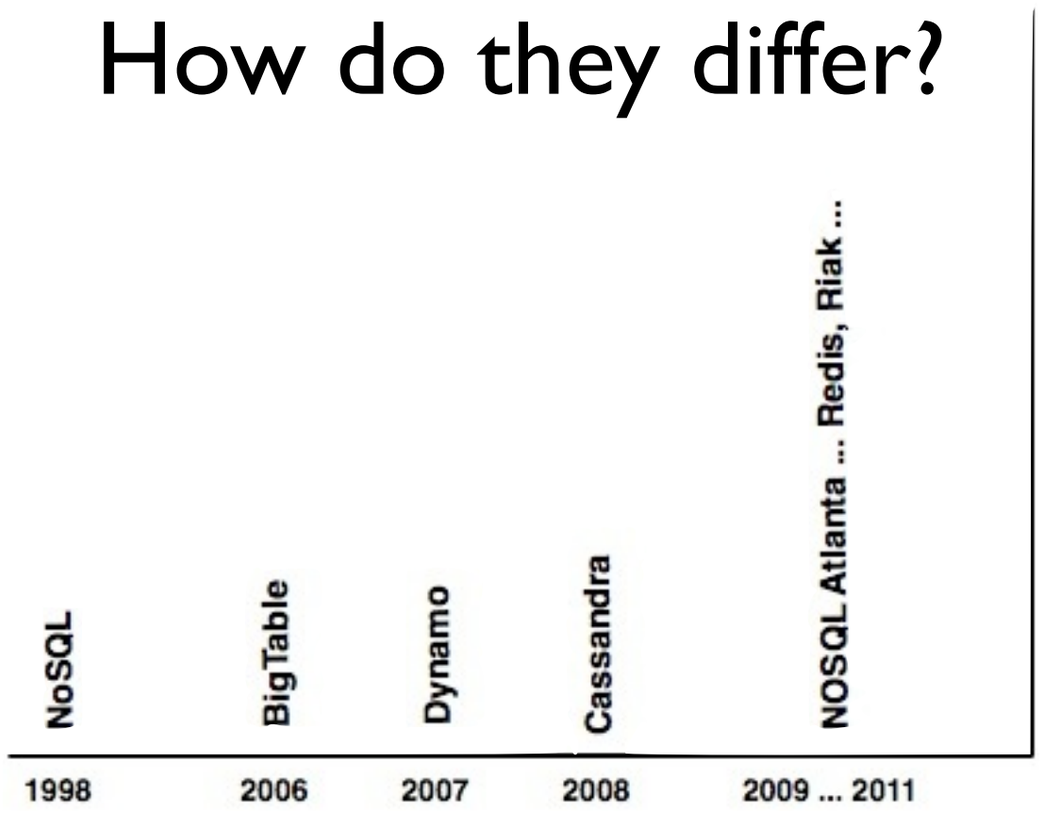
Act II

Non-relational databases
Love is in the air

Non-relational databases

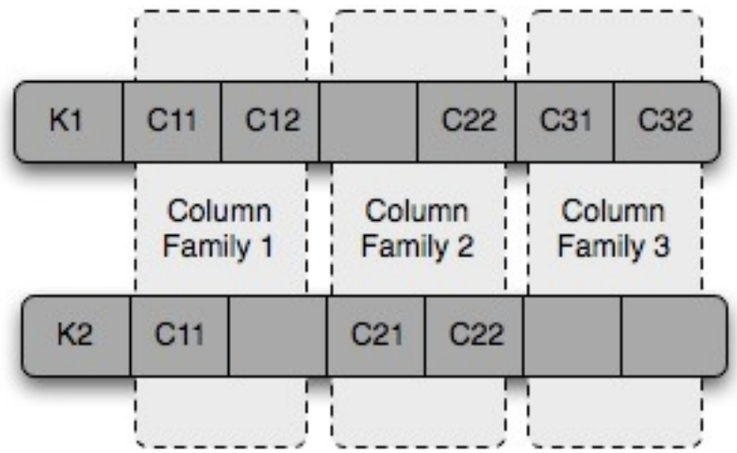
From the origins to the current explosion...

How do they differ?



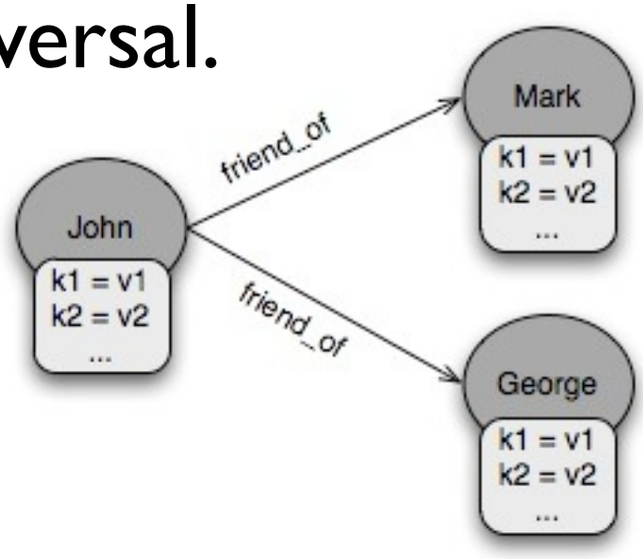
Data Model (1)

- **Column-family**.
  - Key-identified rows
    with a sparse number of columns.
  - Columns grouped in families.
- Multiple families for the same key.
- Dynamically add/remove columns.
- Efficiently access same-family columns



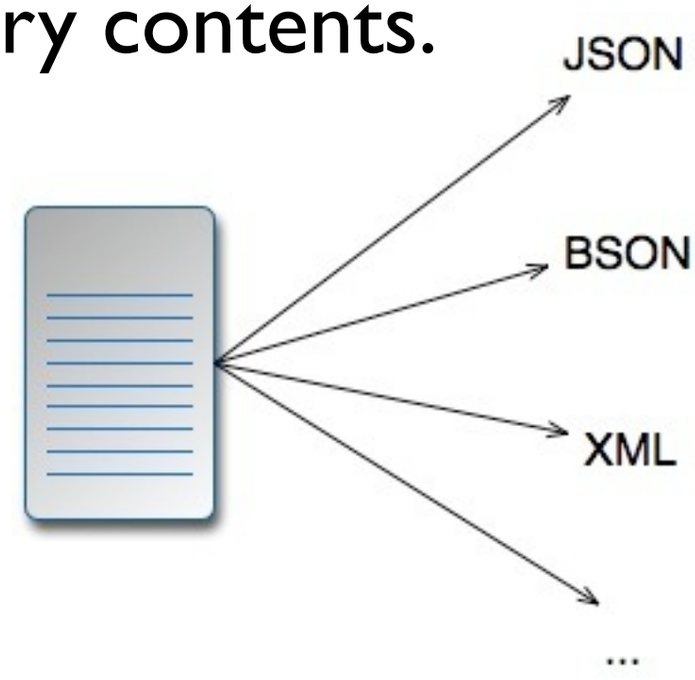
Data Model (2)

- **Graph**.
  - Vertices represent your data.
  - Edges represent meaningful relations between nodes.
  - Key/Value properties attached to both.
  - Indexed properties.
  - Efficient traversal.



Data Model (3)

- **Document**.
  - Schemaless documents.
    - With denormalized data.
  - Stored as a whole unit.
  - Clients can update/query contents.

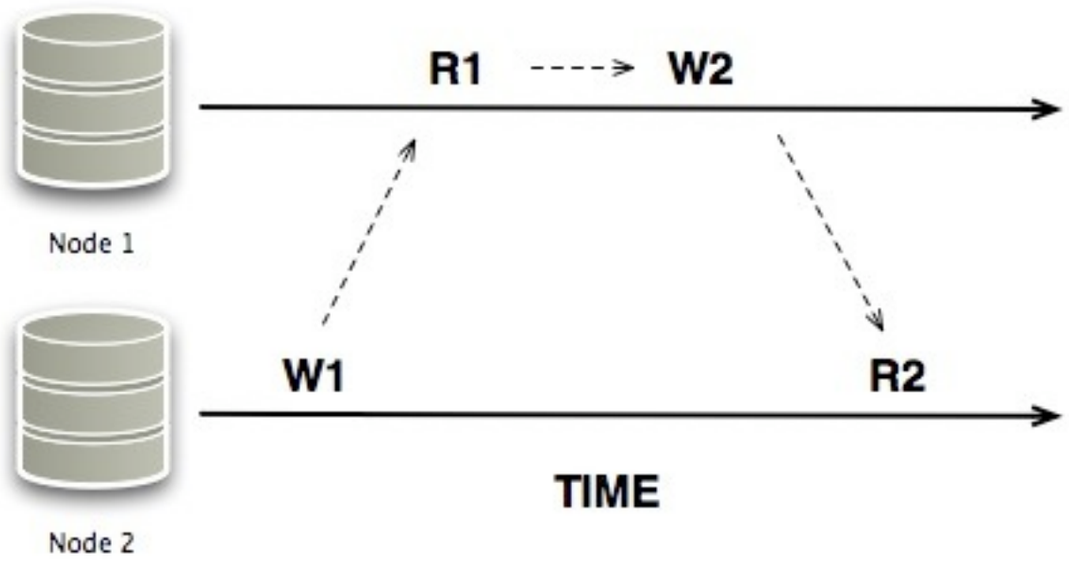


Data Model (4)

- **Key/Value**.
  - Opaque values.
  - Maximize flexibility.
  - Efficiently store and retrieve by key.

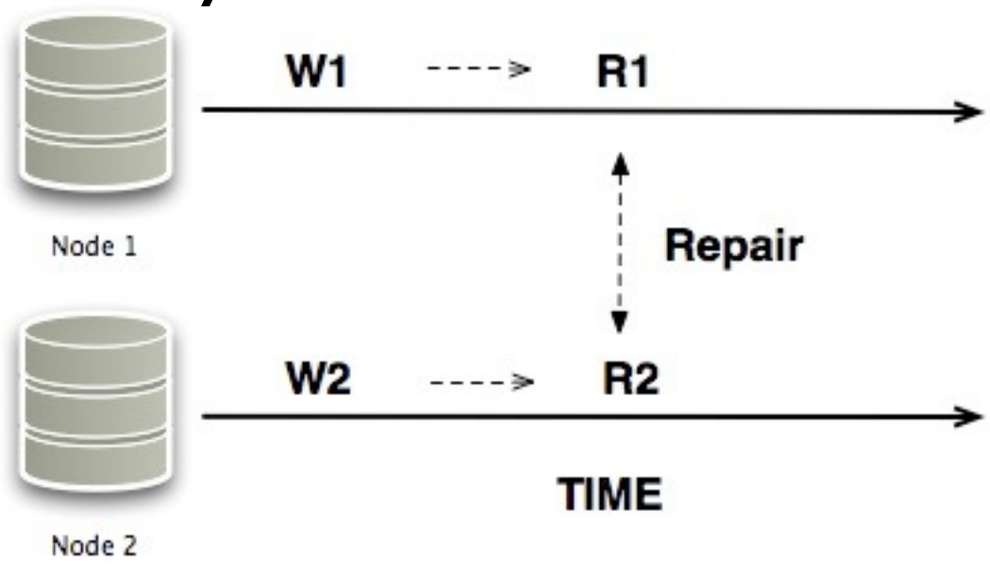
Consistency Model (1)

- **Strict (Sequential) Consistency**.
  - Every read/write operation act on either:
    - The last value read.
    - The last value written.



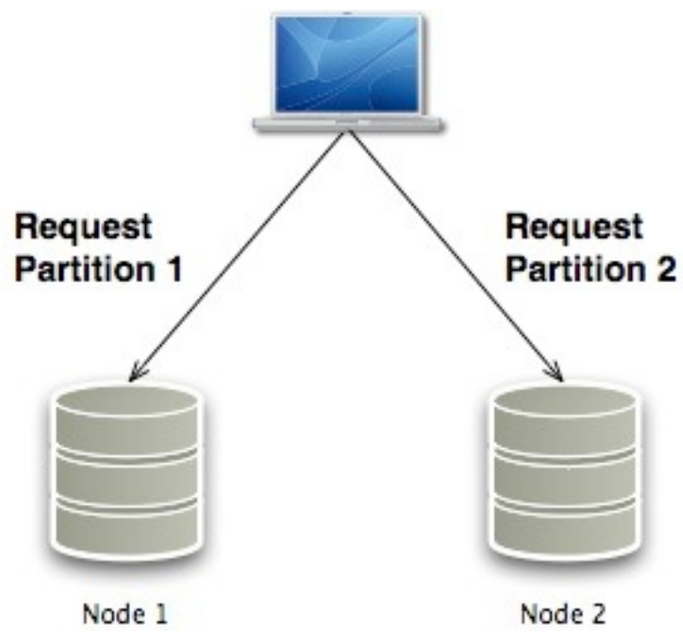
Consistency Model (2)

- **Eventual Consistency**.
  - All read/write operations will eventually reach consistent state.
    - Stale data may be served.
    - Versions may diverge.
    - Repair may be needed.



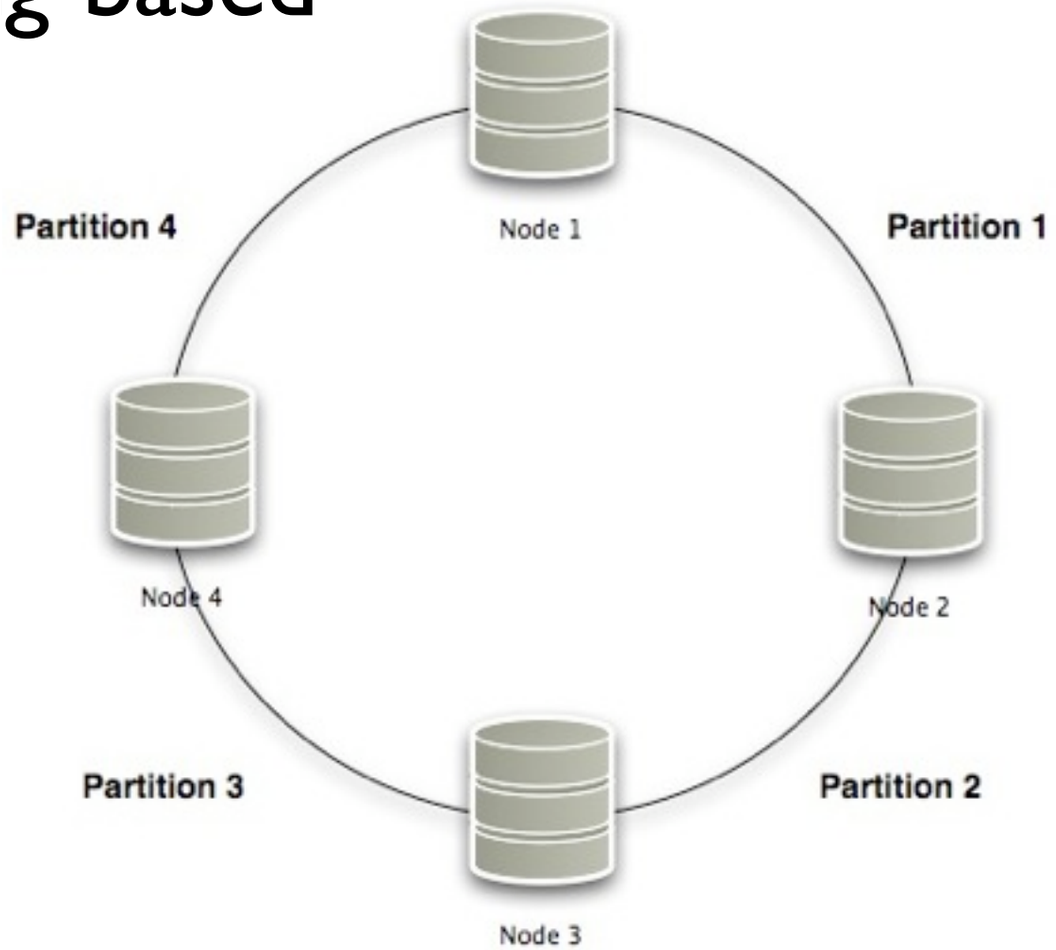
Partitioning (1)

- **Client-side partitioning.**
  - Every server is self-contained.
  - Clients partition data per-request.



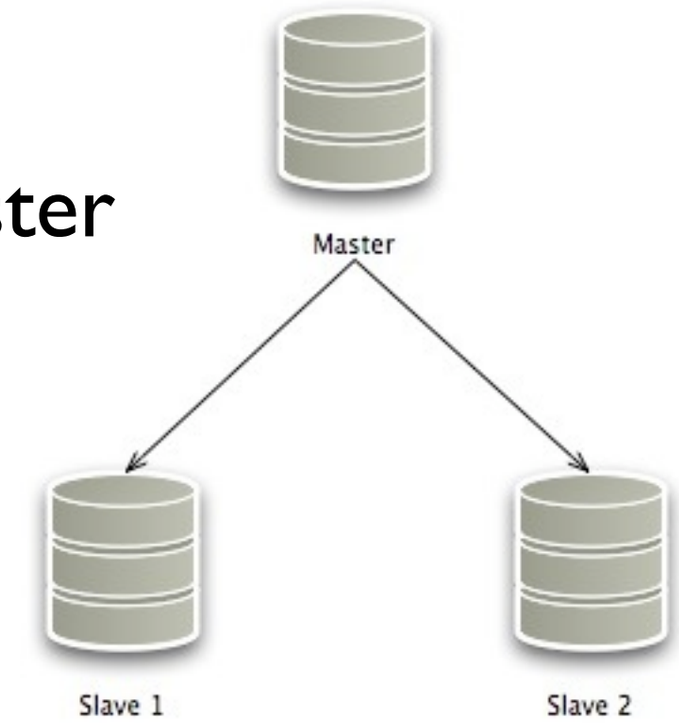
Partitioning (2)

- **Server-side partitioning.**
  - Servers automatically partition data.
  - Consistent-hashing, ring-based is the most used.



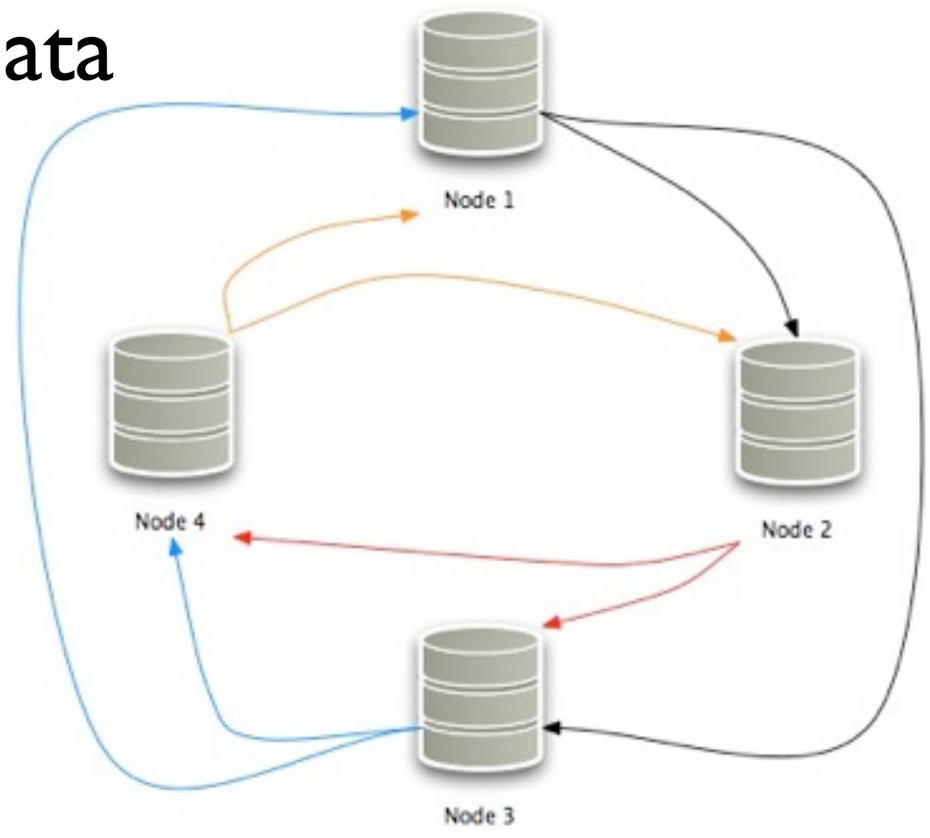
Replication (1)

- **Master/Slave.**
  - Master propagates changes to slaves.
    - Log replication.
    - Statement replication.
  - Slaves may take over master in case of failures.



Replication (2)

- **N-Replication.**
  - Nodes are all peers.
    - No master, no slaves.
  - Each node replicates its data to a subset (N) of nodes.

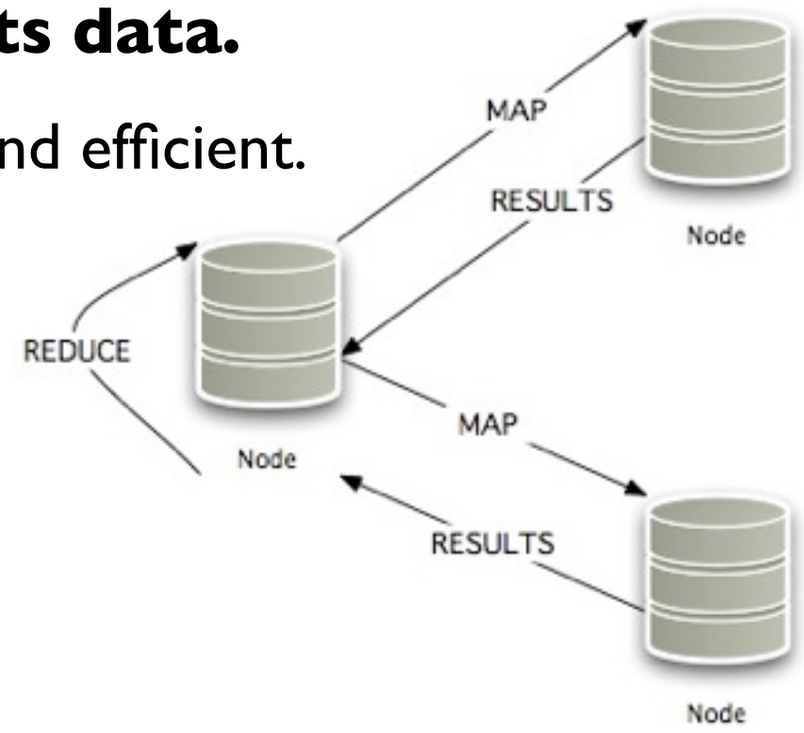


Processing

- A distributed system is built by:
  - Moving data toward its behavior.

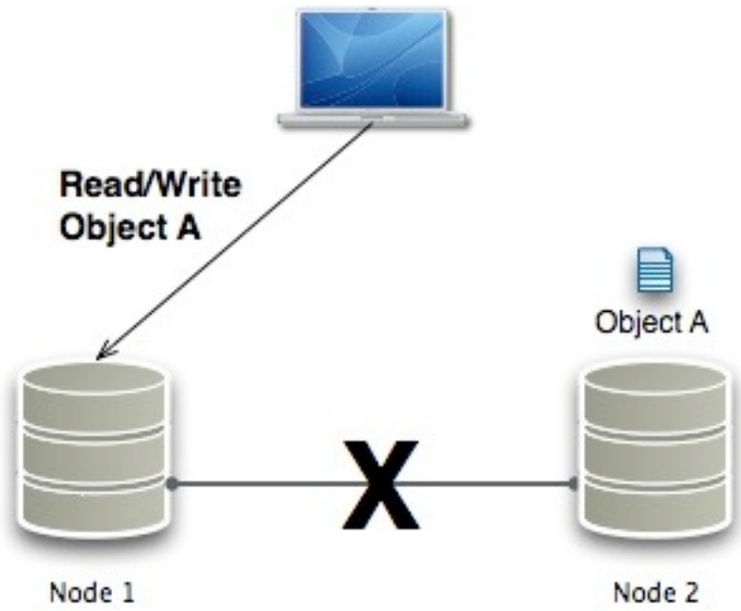
... or ...

  - Moving behavior toward its data.
- An efficient distributed system is built by:
  - **Moving behavior toward its data.**
- Map/Reduce is the most common and efficient.



CAP - Problem

- **CAP Theorem.**
  - Consistency.
  - Availability.
  - Partition tolerance.
  - Pick two.
  - Do you remember?
- Makes sense only
  when dealing with partitions/failures ...



CAP - Trade-offs

- **Consistency + Availability.**
  - Requests will wait until partitions heal.
    - Full consistency.
    - Full availability.
    - Max latency.
- **Consistency + Partition tolerance.**
  - Some requests will act on partial data.
  - Some requests will be refused.
    - Full consistency.
    - Reduced availability.
    - Min latency.
- **Availability + Partition tolerance.**
  - All requests will be fulfilled.
    - Sacrifice consistency.
    - Max availability.
    - Min latency.

Interlude II

Relational or non-relational?
Not the correct question!

Relational or non-relational? Not the correct question!

- Freedom to build the right solution for your problems.
- Freedom to scale your solution as your problems scale.
- Know your use case.
  - Understand the problem domain, then choose technology.
- Know your data.
  - Understand your data and data access patterns, then choose technology.
- Know your tools.
  - Understand available tools, don't go blind.
- Pick the simple solution.
  - Choose the simpler technology that works for your problem.
- Build on that.

Act III

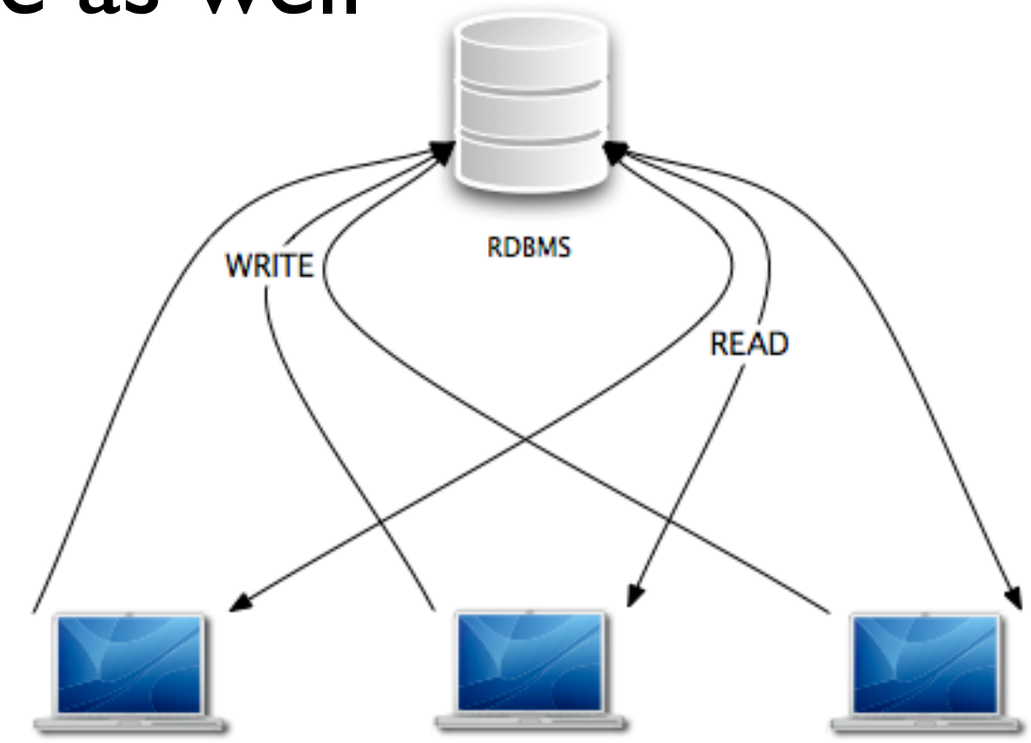
Building scalable apps

Building scalable apps

Relational databases

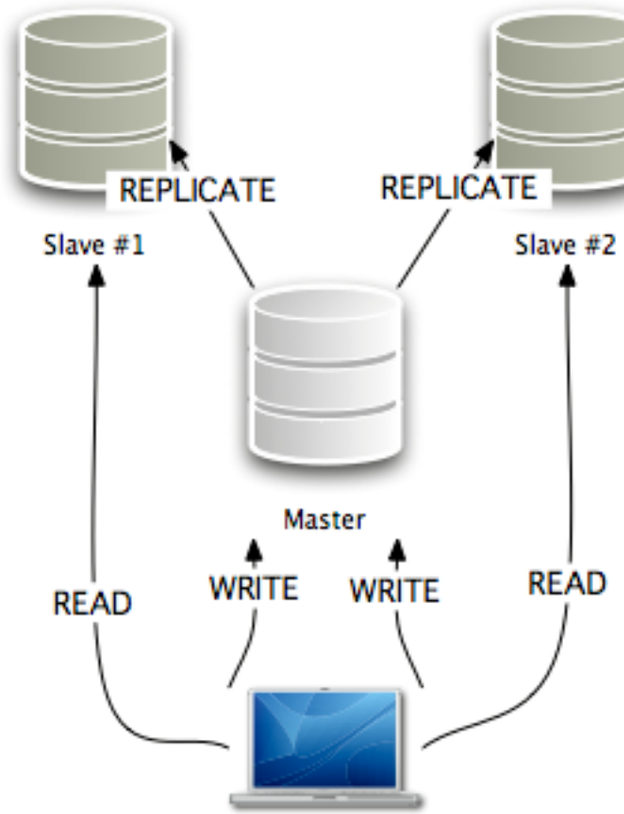
Scaling out...

- Adding app servers will work...
  ... for a little while!
- But at some point we need to scale the database as well



Master-Slave

- One master
  - gets all the writes
  - replicates to slaves
- Slaves (& master)
  - gets the read operations

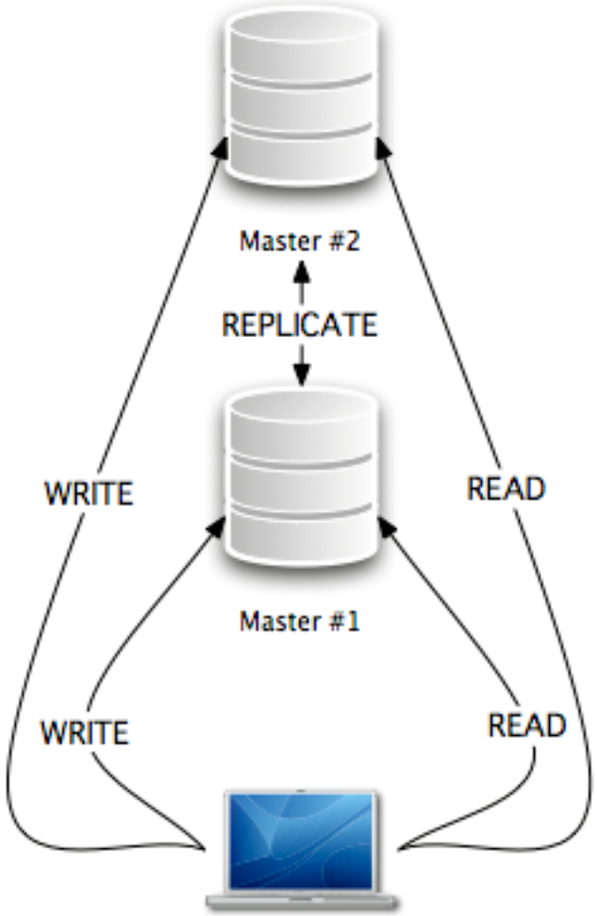


- We didn't really scale writes
- Static topology
- SPOF remains

Master-Master

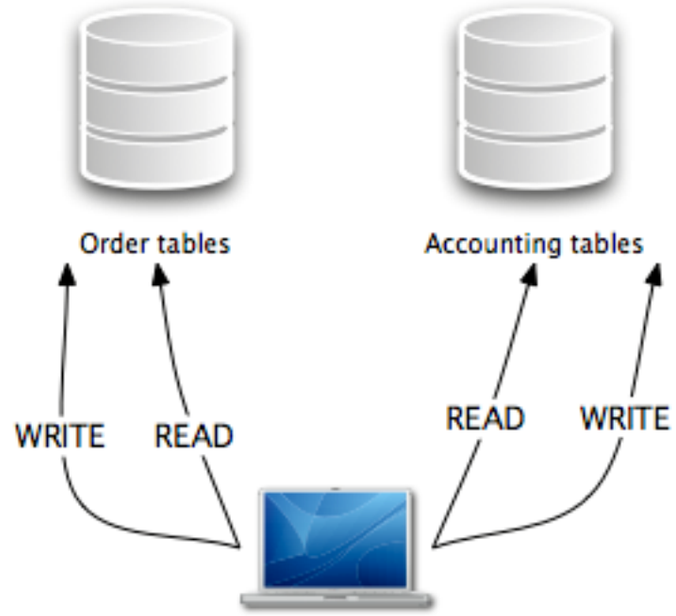
- Multiple masters
  - Writes & reads all participants
- Writes are replicated to masters
- Synchronously
  - Expensive 2PC
- Asynchronously
  - Conflicts have to be resolved

- Complex
- Static topology
- Limited performance again
- Solved SPOF, sort of…



Vertical Partitioning

- Data is split across multiple database servers based on
  - The functional area
- Joins are moved to the application
  - Not relational anymore
- SPOF back
- What about one functional area growing "out of hand" ?

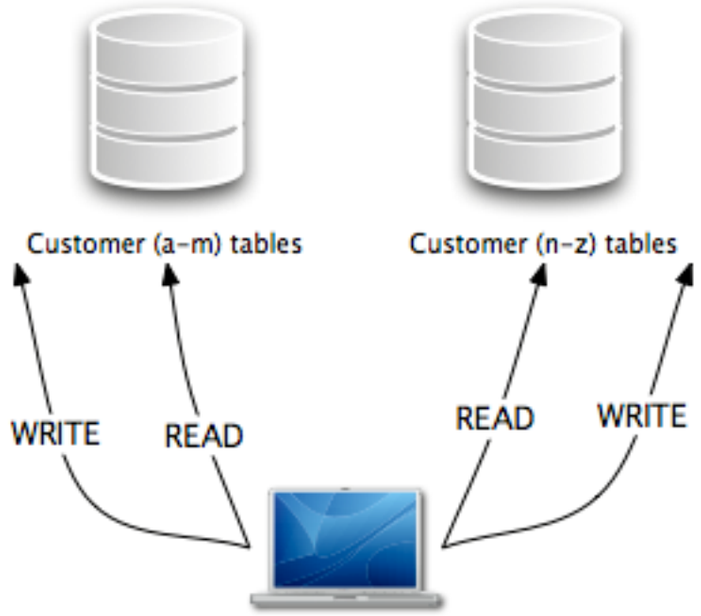


Horizontal partitioning

- Data is split across multiple database servers based on
  - Key sharding

- Joins are moved to the application
  - Not relational anymore

- SPOF back
- What about one functional area growing "out of hand" ?
- Routing required
  - Where's Order#123 ?



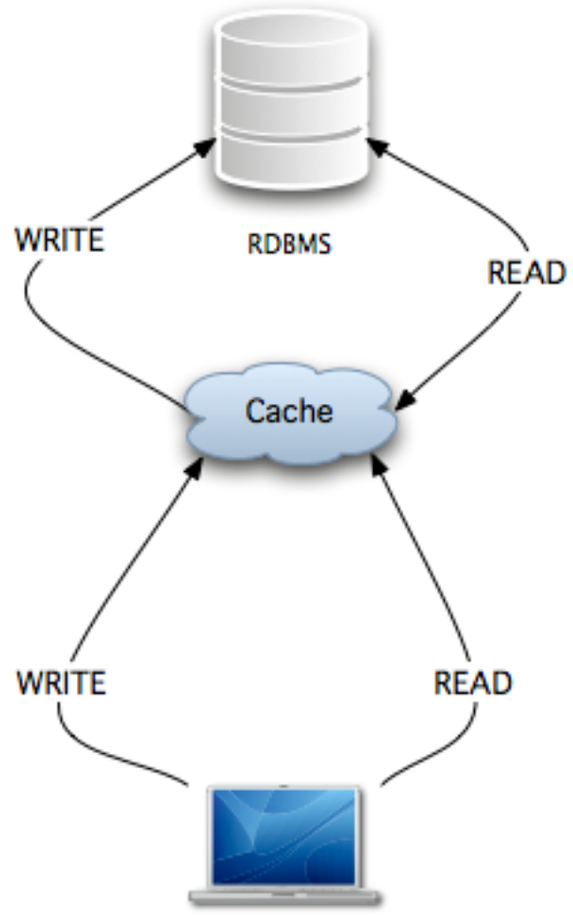
Building scalable apps

Caching

Caching

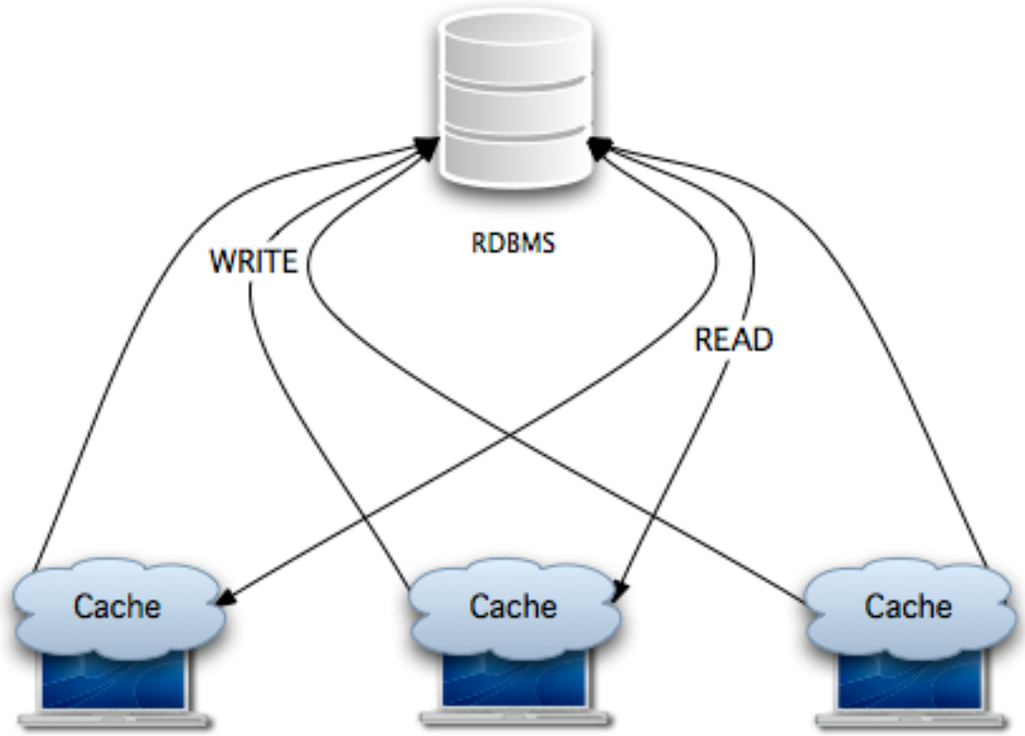
- If going to the database is so expensive...
  ... we should just avoid it!
- Put a cache in front of the database

- Data remains closer to processing unit
- If needed, distribute



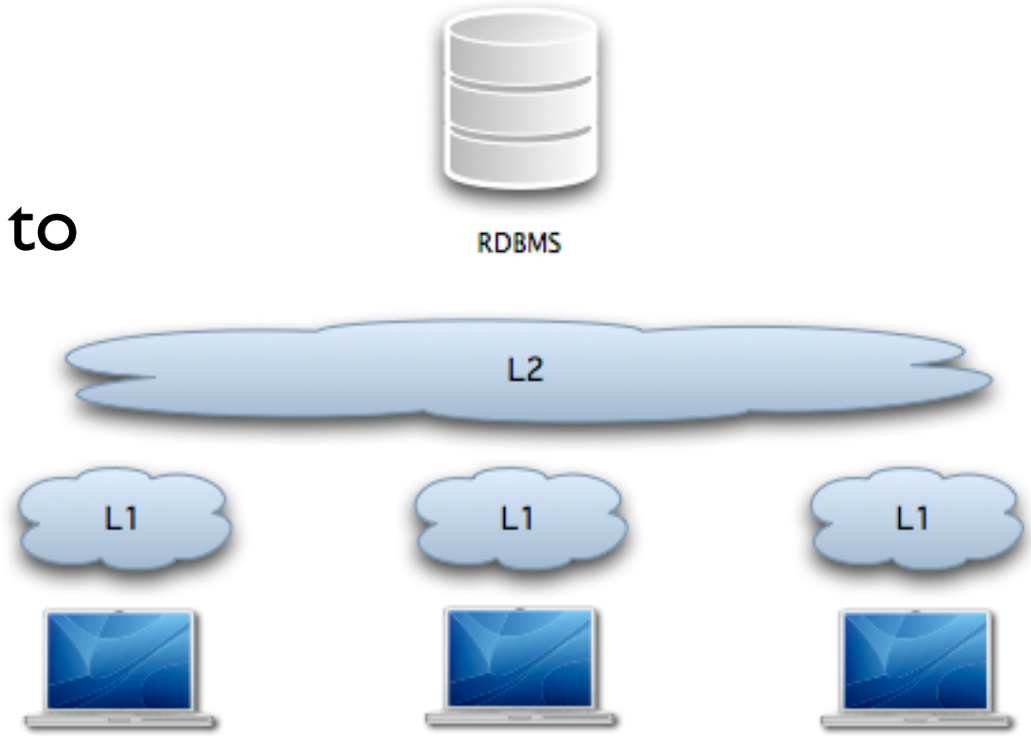
Distributed Caching

- What if data isn't perfectly partitioned ?
- How do we keep this all in sync ?
- Peer-to-peer ?

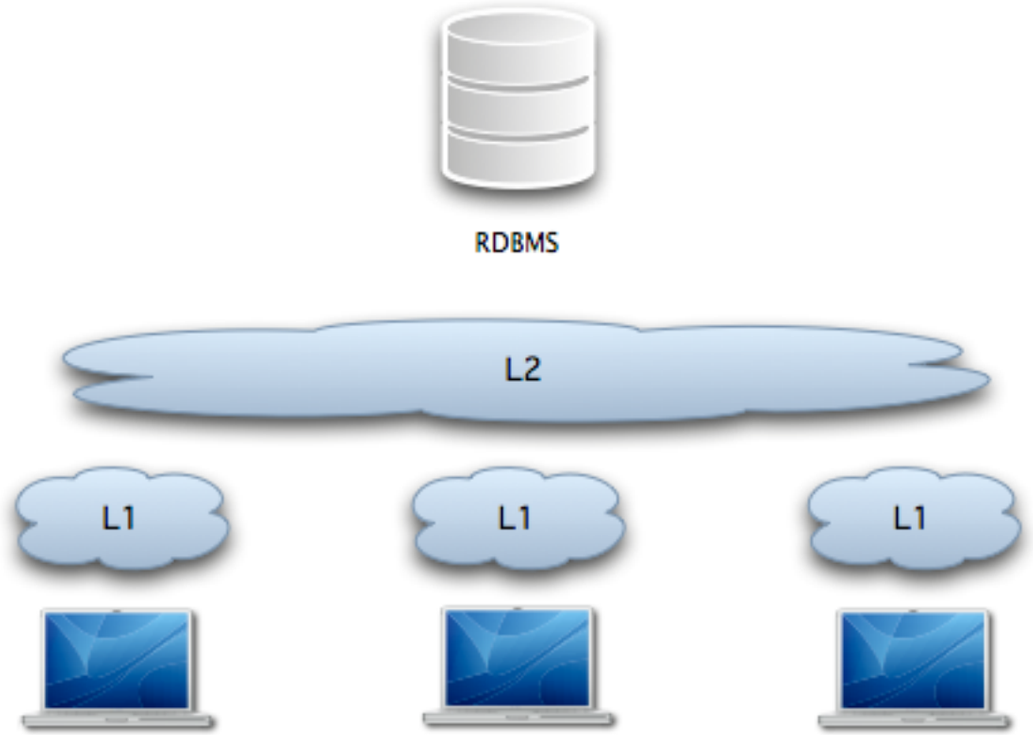


Distributed Caching

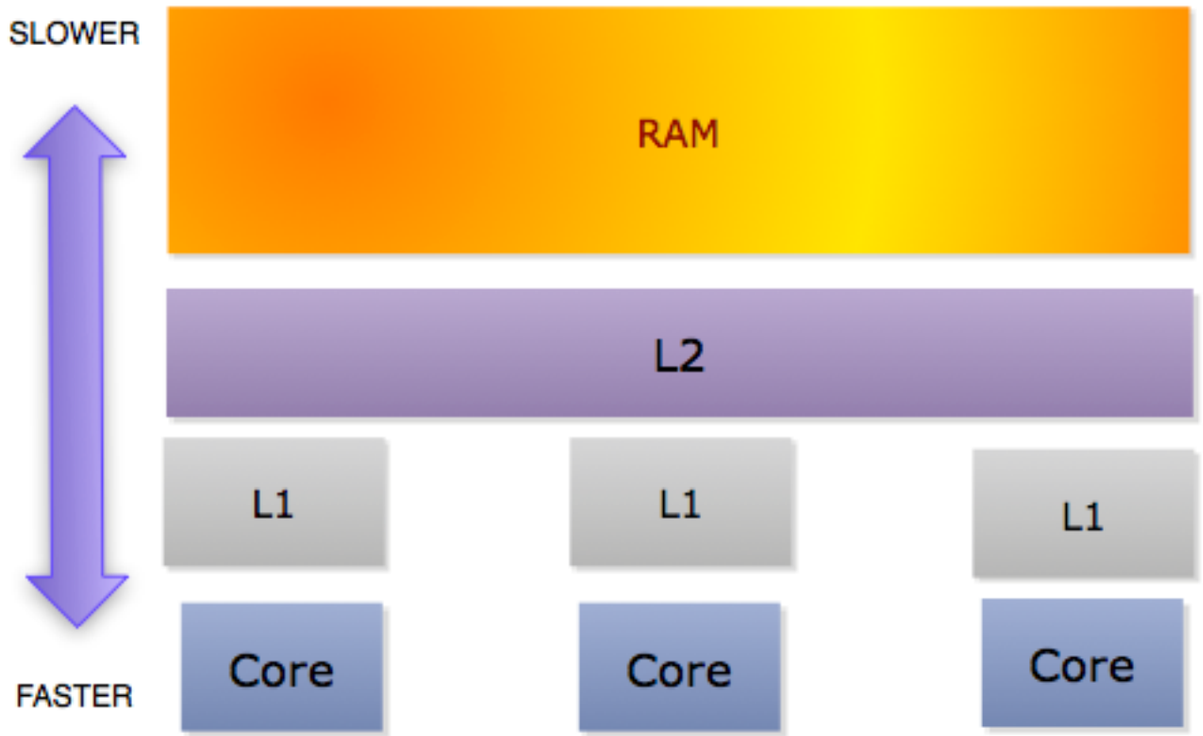
- Cached data remains close to the processing unit
- Central unit orchestrates it all



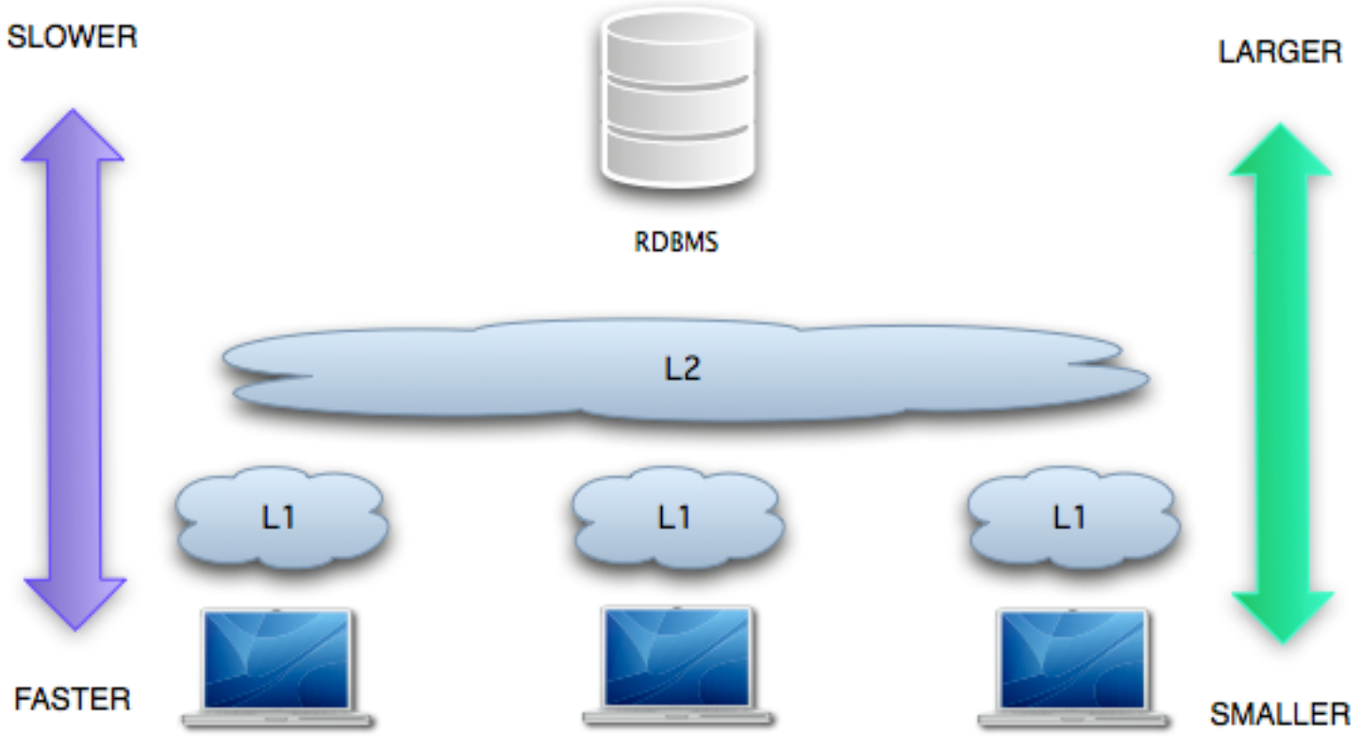
Distributed Caching



Distributed Caching

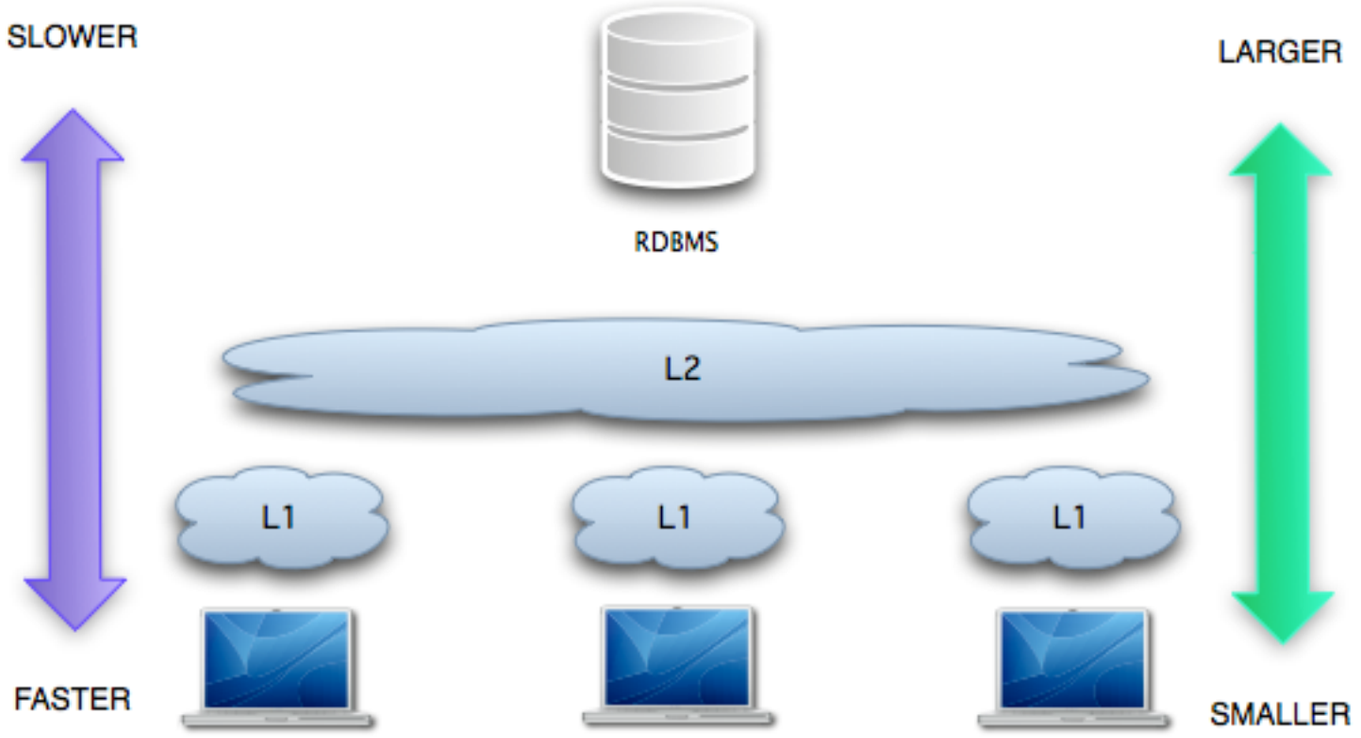


Distributed Caching



Distributed Caching

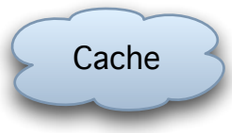
We've scaled our reads
But what about writes ?



Write-behind Cache

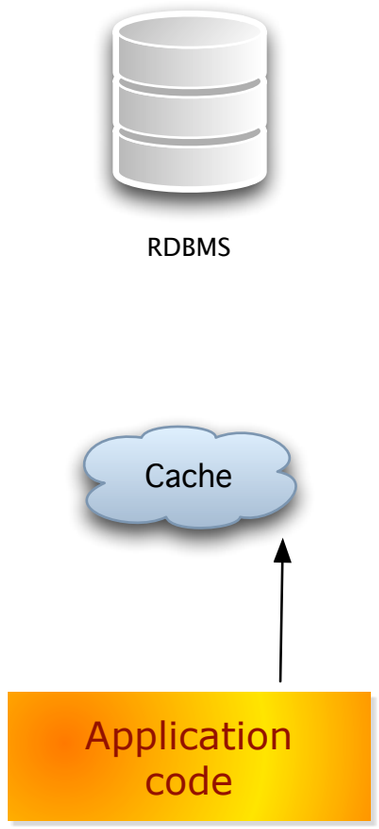
- Rather than write changes directly to the slowest participant
  - Write to fastest durable store (persistent queue)
    - required for recovery in the face of failure
  - Only write to database later
    - in batches and/or coalesced
    - while still controlling
      - the lag
      - the load on the database
- In a distributed environment handling failures we enforce happens at least once
  - loosens the contract vs. "once and only once"!

RDBMS

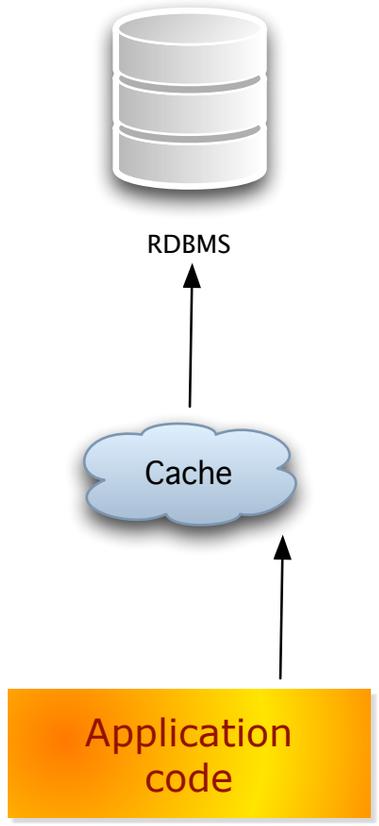


Application code

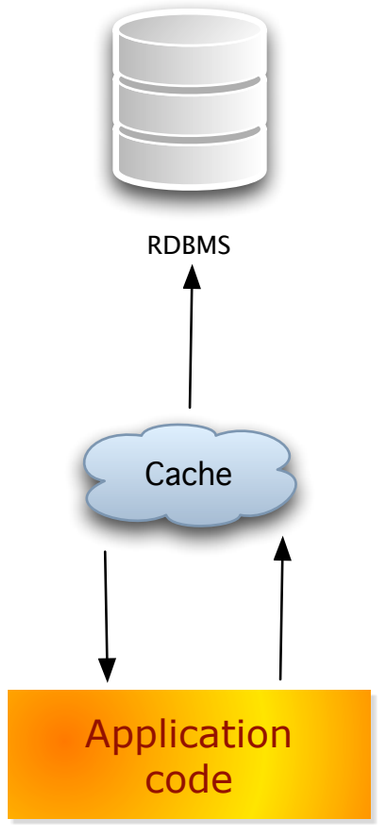
Write-through



Write-through

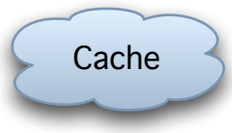


Write-through



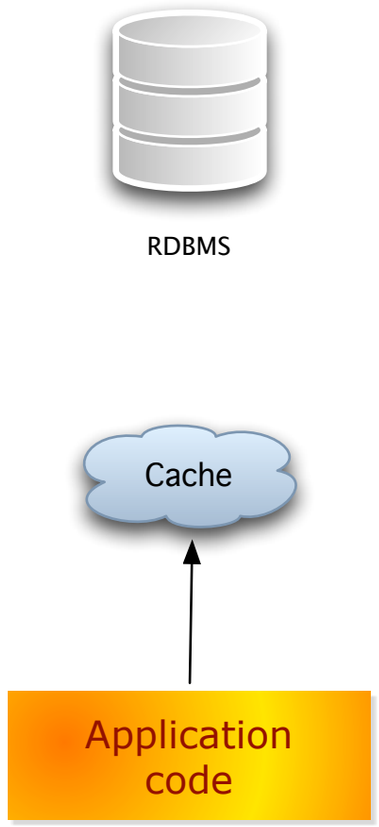
Write-through

RDBMS

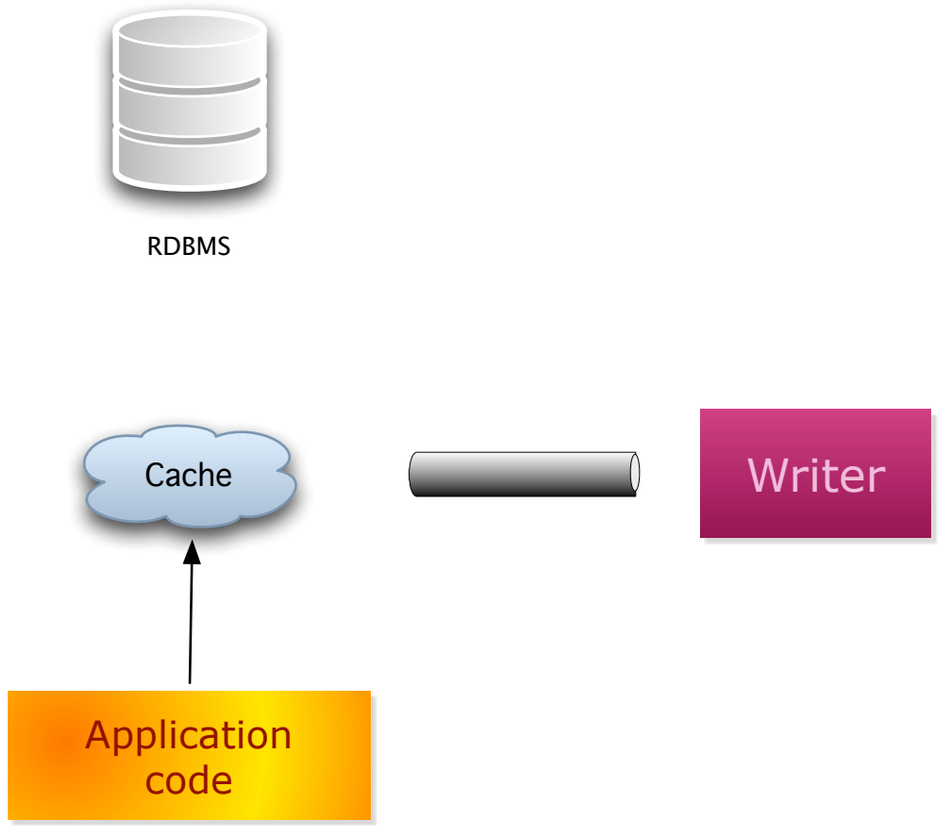


Application code

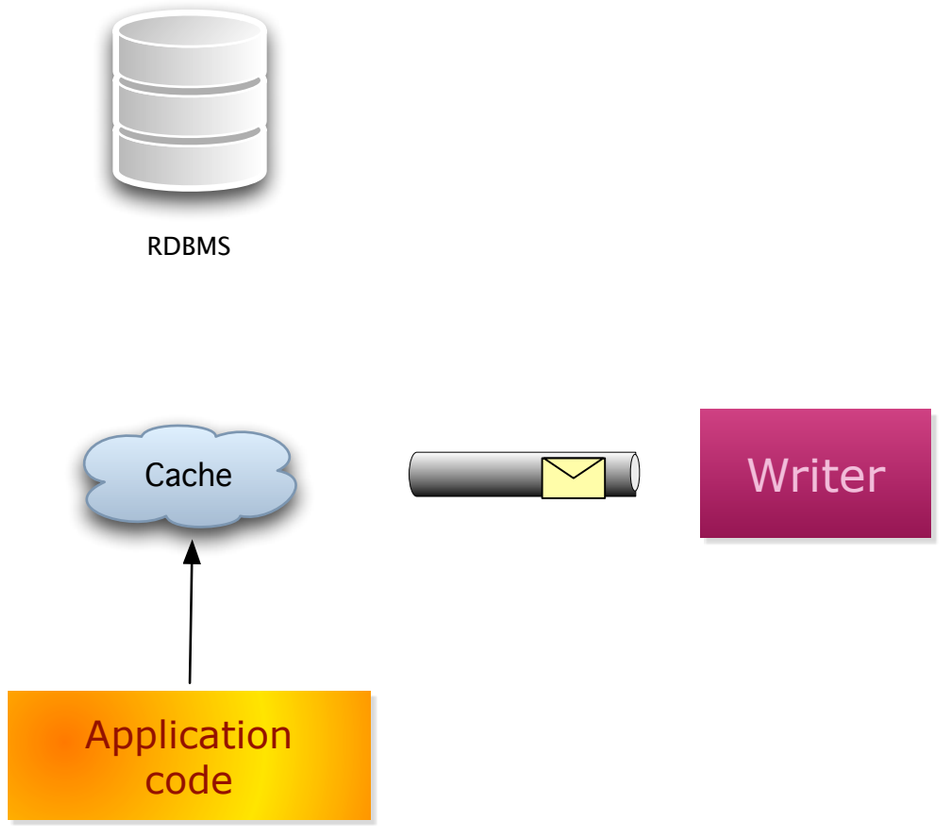
Write-behind



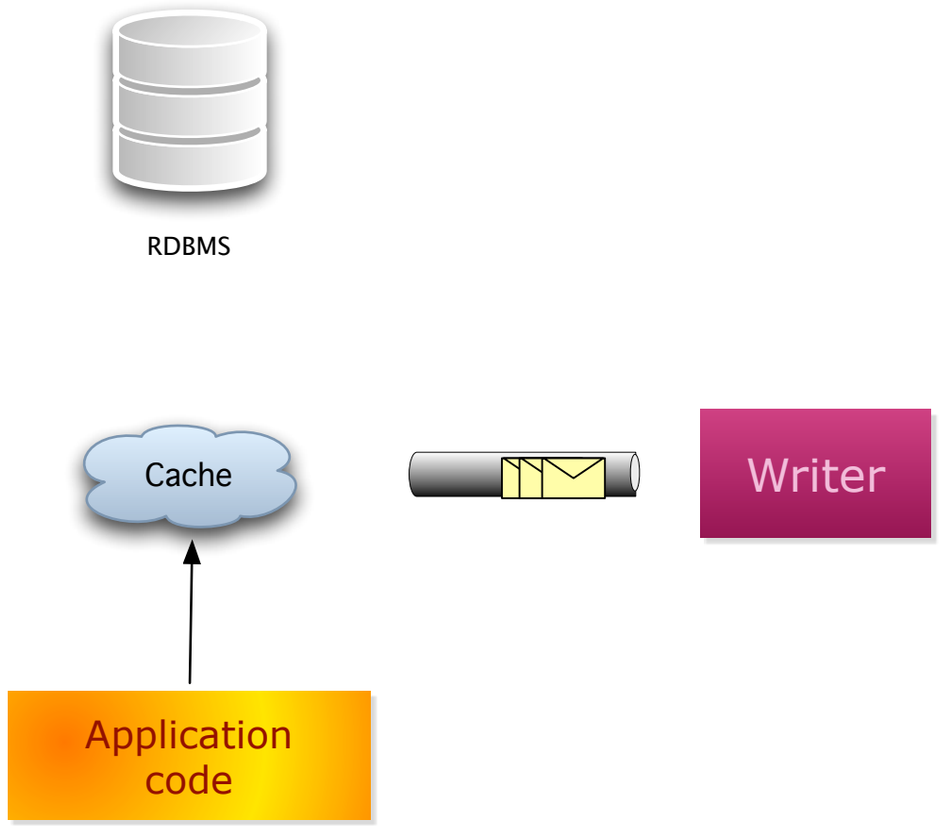
Write-behind



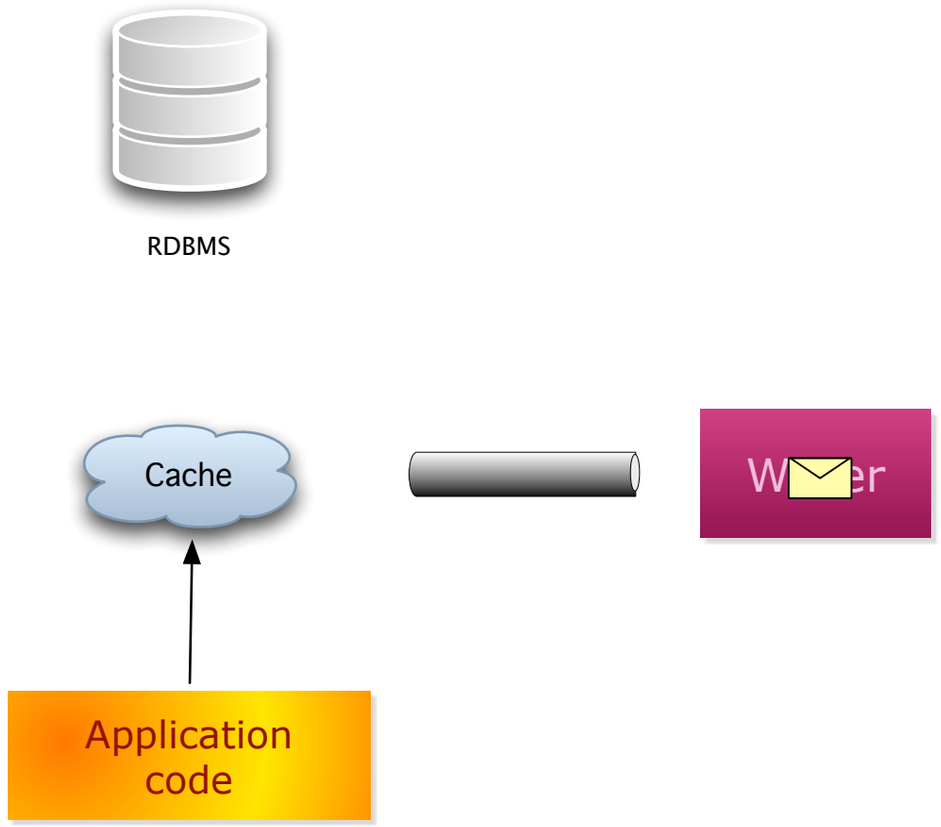
Write-behind



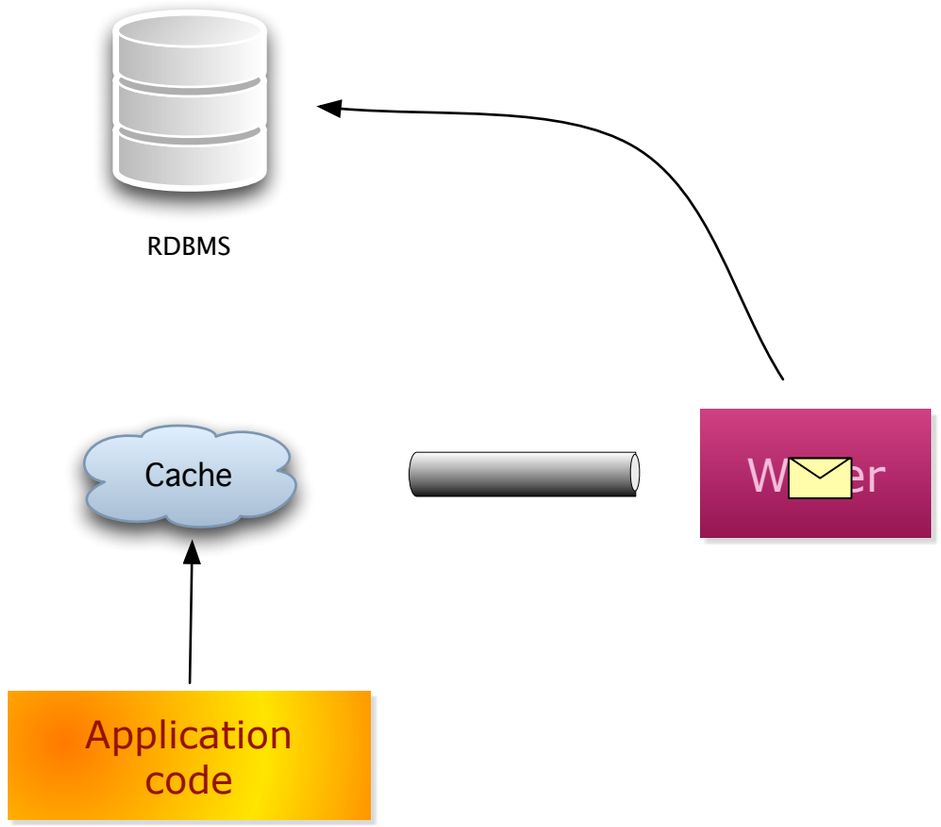
Write-behind



Write-behind



Write-behind



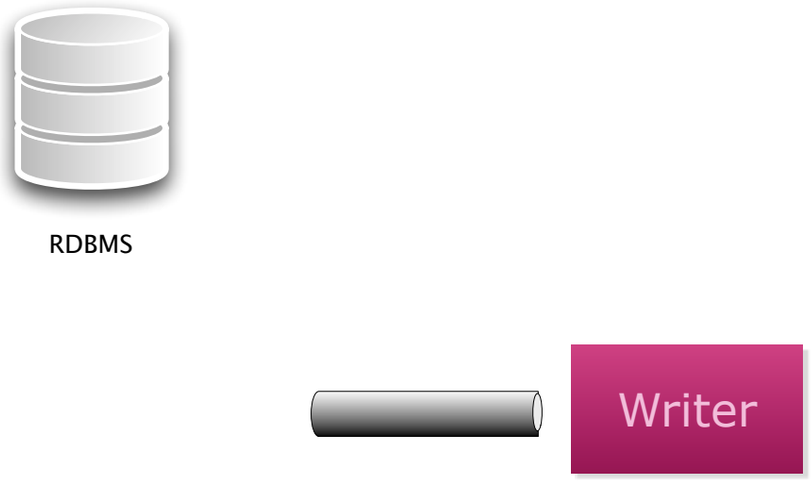
Write-behind

RDBMS

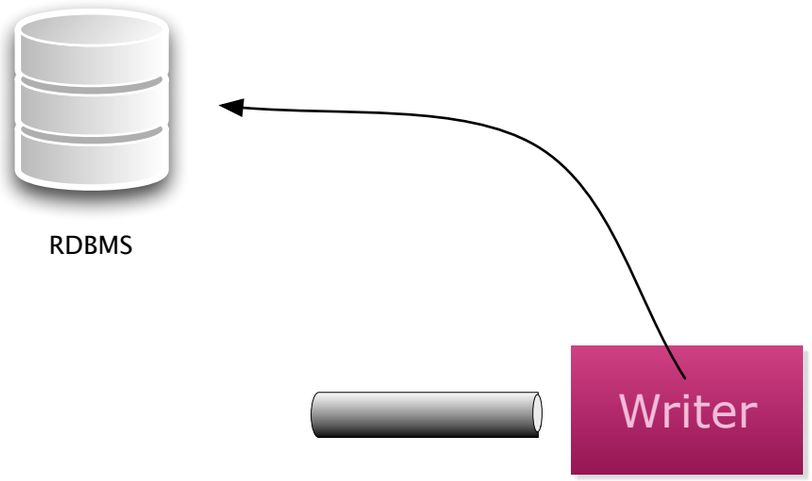
Write-behind

RDBMS

Write-behind



Write-behind



Write-behind

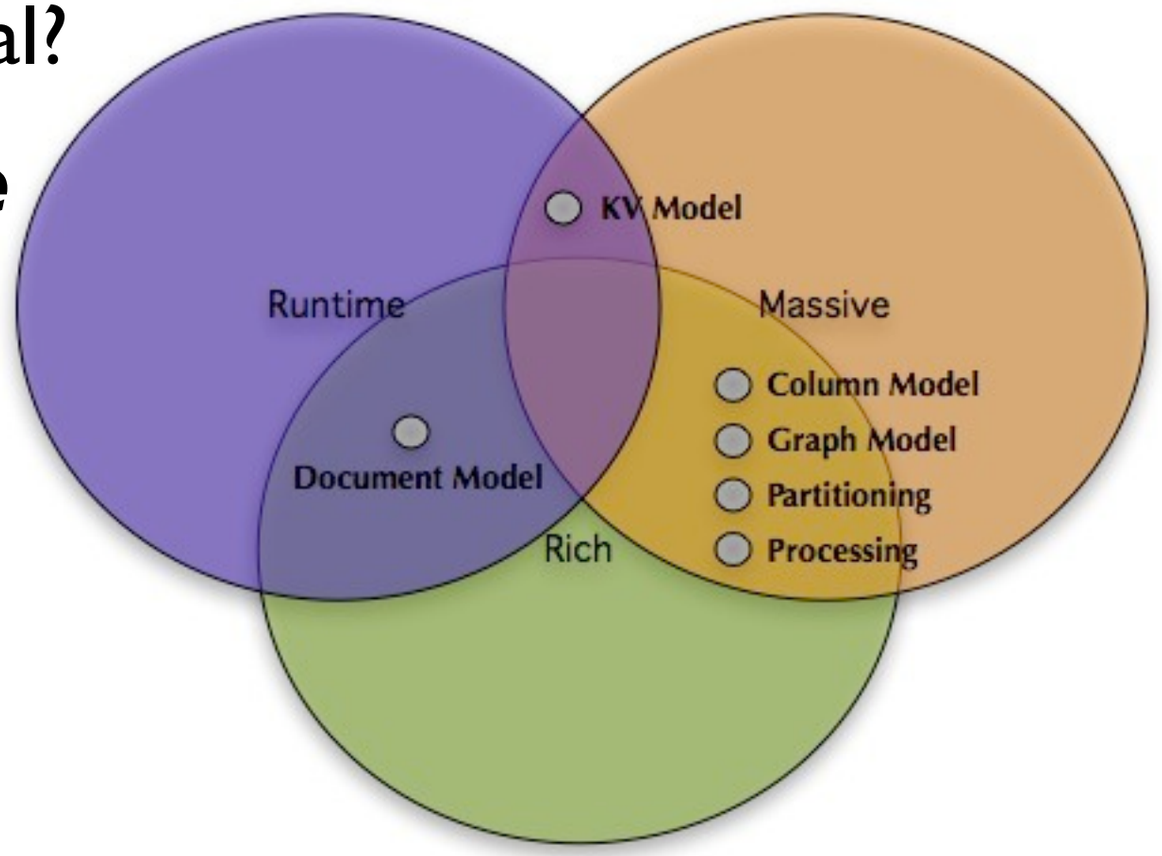
Building scalable apps

Non-relational databases

The case for non-relationals

- We've seen how to scale our relational database.
- We've seen how to add caching.
- We've seen how to make caching scale.

- So why going non-relational?
  - ***A matter of use case***



Rich Data

- **Frequent schema changes.**
  - Relational databases
    cannot easily handle table modifications.
  - Column, Document and Graph databases can.
- **Tracking/Processing of large relations.**
  - Relational databases keep and
    traverse table relations by foreign-key joins.
    - Joins are expensive.
  - Graph databases provide cheap and
    fast traversal operations.

Runtime Data

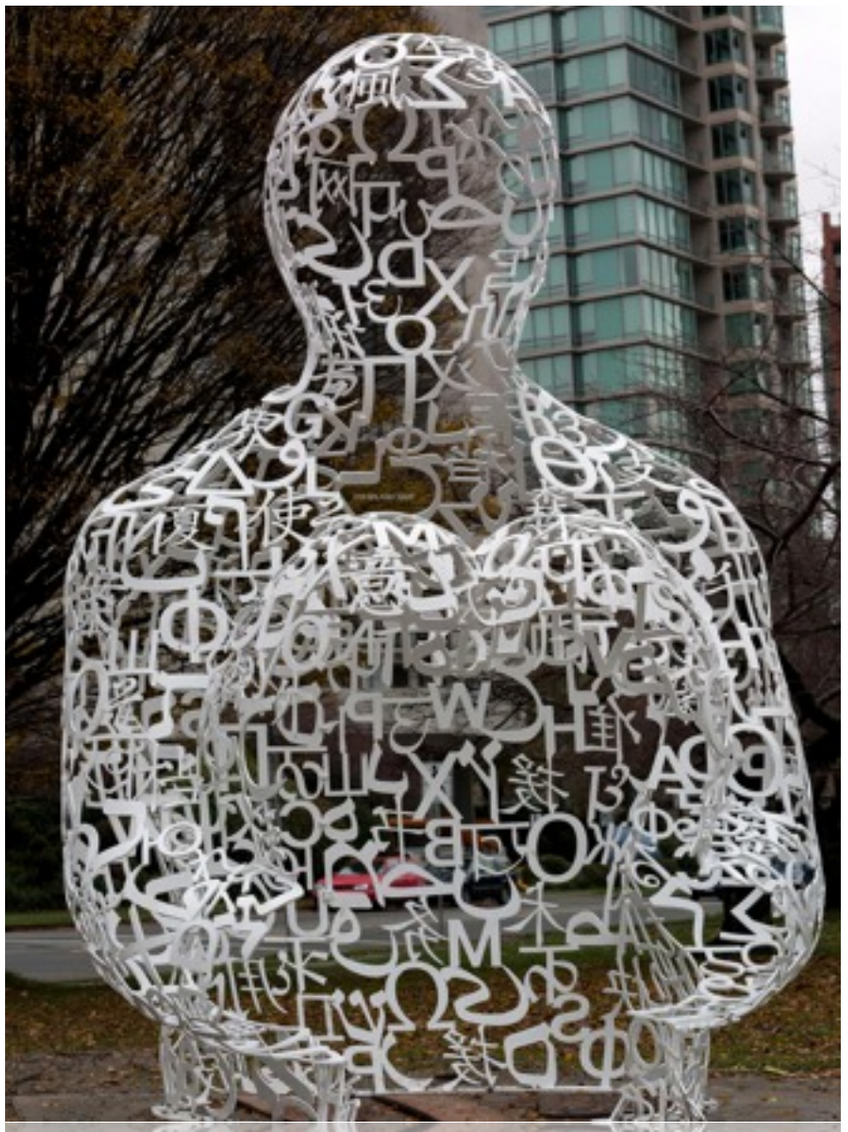
- **Poorly structured data.**
  - Relational model
    doesn't fit unstructured data.
    - Unless you want BLOBs.
  - Non-relational databases provide more flexible models.
- **Throw-away data.**
  - Relational databases provide maximum data durability.
  - But not all kind of data are the same.
  - When you can afford to possibly lose some data, non-relational databases let you trade durability for higher flexibility and performance.

Massive Data

- **Large quantity of data.**
  - Relational databases are typically single instance.
    - Otherwise cost too much.
  - Choose a partitioned non-relational database.
- **High volume of writes.**
  - Scaling writes with relational databases is difficult.
    - Even if using write-behind caching.
  - Choose a partitioned non-relational database.
  - Eventual consistency may also help.
- **Complex, aggregated, queries.**
  - Complex aggregations and queries
    are expensive in standard relational databases.
    - Remember joins?
  - Choose a non-relational database supporting distributed data processing.
    - Hint: Map/Reduce.

Building scalable apps

Multi-paradigm



Creative Commons by Eyesplash

A case for multi-paradigm: CQRS

CQRS - Explained

- **Command Store.**
  - Strictly modeled after the problem domain.
  - Commands model domain changes.
  - Domain changes cause events publishing.
- **Query Store.**
  - Fed by published events.
  - Strictly modeled after the user interface.
  - Possibly denormalized to accommodate queries.
- **Offline Store.**
  - Fed by published events.
  - Strictly modeled after processing needs.
    - Business Intelligence.
    - Reporting.
    - Statistical aggregations, …

What about stores implementation?

Introducing Ehcache

- Started in 2003 by Greg Luck
- Apache 2.0 License
- Integrated by lots of projects, products
- Hibernate Provider implemented 2003
- Web Caching 2004
- Distributed Caching 2006
- REST and SOAP APIs 2008
- Acquired by Terracotta Sept. 2009

Introducing Ehcache 2.4

- New Search API
- New consistency modes
- New transactional modes

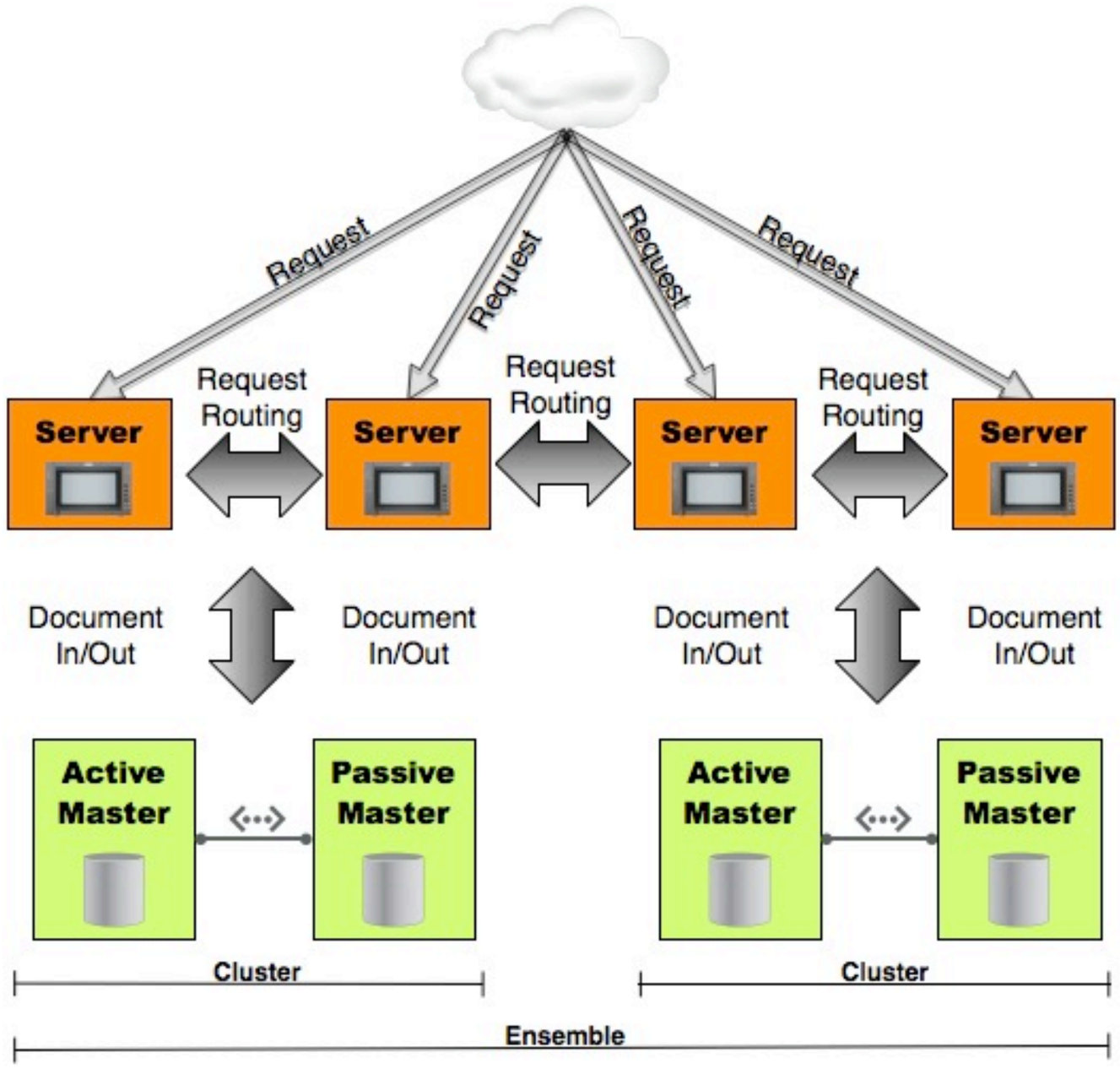
- Still:
  - Small memory footprint
  - Cache Writers
  - JTA
  - Bulk load
- Grows with your application
  with only two lines of configuration
  - Scale Up - BigMemory (100's of Gig, in process, NO GC)
  - Scale Out - Clustering Platform (Up to 2 Terabytes, HA)

Ehcache & Terracotta

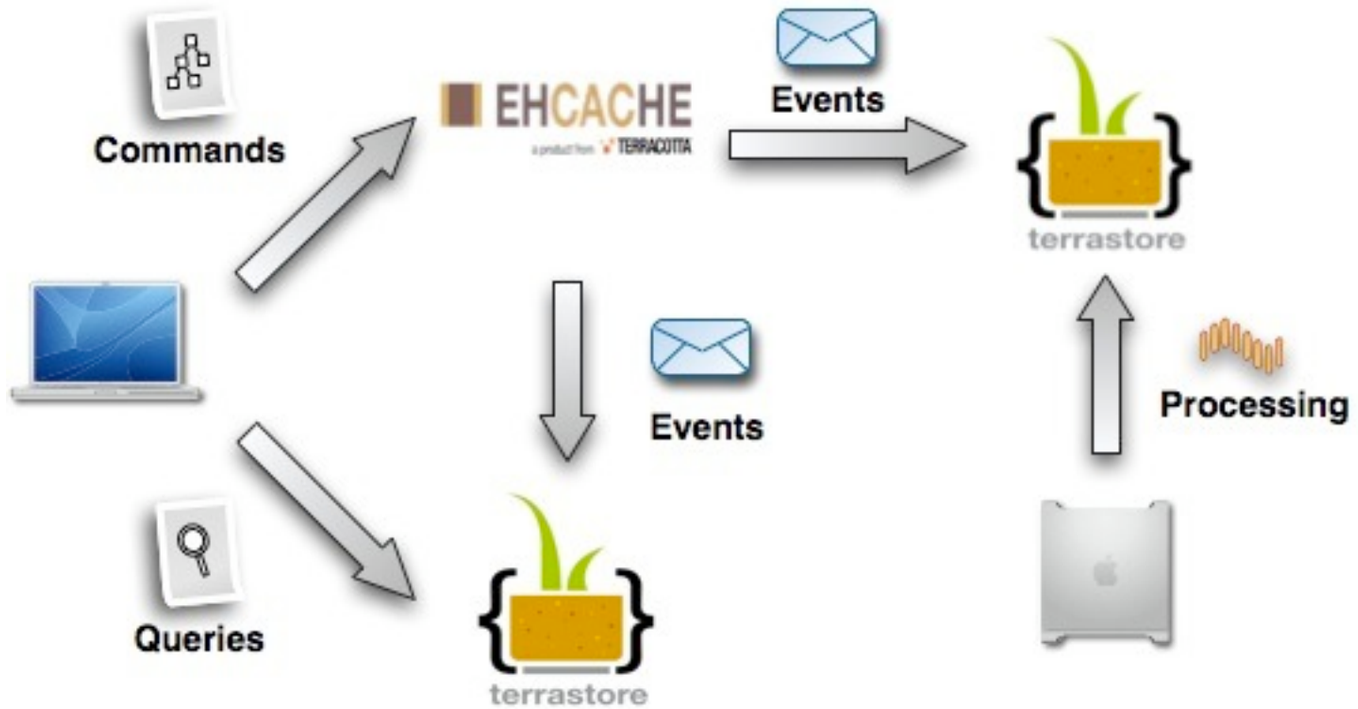
- Terabyte scale with minimal footprint
- Server array with striping for linear scale
- High availability
- High performance data persistence
- In-memory performance as you scale

Introducing Terrastore

- Document Store.
  - Ubiquitous.
  - Consistent.
  - Distributed.
  - Scalable.
- Written in Java.
- Based on Terracotta.
- Open Source.
- Apache-licensed.



Terrastore Architecture



CQRS - Implemented

CQRS - Implemented

- **Command Store.**
  - Express your domain as an object model.
  - Process and store it with Ehcache and Terracotta.
  - Optionally write it back to a relational database.
- **Query Store.**
  - Map queries to user (screen) views.
  - Map views to JSON documents.
  - Store and get them back with Terrastore.
- **Offline Store.**
  - Collect data from input commands.
  - Keep data denormalized.
  - Aggregate and process with Terrastore Map/Reduce.

The end

Be a polyglot!

Conclusion

- Be polyglot
  - Go out and play with all this
  - Know your options (all of them)
- Understand your domain
  - It's current and future requirements
- Be wise, and choose well!

More info

- www.ehcache.org
- www.terracotta.org
- code.google.com/p/terrastore
  - Lightning talk today at 15:35 in room A4
- Visit us on the booth