Carnegie Mellon University

26 Final Review + Systems Potpourri

Intro to Database Systems
15-445/15-645
Fall 2020

Andy Pavlo
Computer Science
Carnegie Mellon University

FINAL EXAM

Who: You

What: Final Exam

Where: Gradescope + OHQueue + Google Doc

When: Thu Dec 17th (Two Sessions)

Why: https://youtu.be/yCotpBAqJho

https://15445.courses.cs.cmu.edu/fall2020/final-guide.html

FINAL EXAM

Two Exam Sessions:

→ Session #1: Thu Dec 17th @ **8:30am ET**
→ Session #2: Thu Dec 17th @ **8:00pm ET**
→ *I will email you to confirm your session.*

Exam will be available on Gradescope.

Please email Andy if you need special accommodations.

FINAL EXAM

Exam covers all lecture material in the entire course but will emphasize topics after mid-term.

Open book/notes/calculator.

You are not required to turn on your video during the video.

We will answer clarification questions via OHQ and post announcements on a Google Doc.

COURSE EVALS

Your feedback is strongly needed:

→ https://cmu.smartevals.com
→ https://www.ugrad.cs.cmu.edu/ta/F20/feedback/

Things that we want feedback on:

→ Homework Assignments
→ Projects
→ Reading Materials
→ Lectures

OFFICE HOURS

Andy's hours:

→ Monday Dec 14th @ 3:20-4:40pm

→ Mon Dec 14th + Wed Dec 16th @ 10pm: https://calendly.com/andy-pavlo/f20-andy-after-dark

→ Or by appointment

All TAs will have their regular office hours up to and including Saturday Dec 12th

STUFF BEFORE MID-TERM

SQL

Buffer Pool Management

Hash Tables

B+Trees

Storage Models

Inter-Query Parallelism

TRANSACTIONS

ACID

Conflict Serializability:
- → How to check?
- → How to ensure?

View Serializability

Recoverable Schedules

Isolation Levels / Anomalies

TRANSACTIONS

Two-Phase Locking
→ Rigorous vs. Non-Rigorous
→ Deadlock Detection & Prevention

Multiple Granularity Locking
→ Intention Locks

TRANSACTIONS

Timestamp Ordering Concurrency Control
→ Thomas Write Rule

Optimistic Concurrency Control
→ Read Phase
→ Validation Phase
→ Write Phase

Multi-Version Concurrency Control
→ Version Storage / Ordering
→ Garbage Collection

CRASH RECOVERY

Buffer Pool Policies:
→ STEAL vs. NO-STEAL
→ FORCE vs. NO-FORCE

Write-Ahead Logging

Logging Schemes

Checkpoints

ARIES Recovery
→ Log Sequence Numbers
→ CLRs

DISTRIBUTED DATABASES

System Architectures

Replication

Partitioning Schemes

Two-Phase Commit

2018		2019		2020	
Cockroach LABS	26	Scuba	20	amazon DynamoDB	18
Google Spanner	25	mongoDB	19	cassandra	15
mongoDB	24	Cockroach LABS	18	mongoDB	14
Amazon Aurora	18	Amazon Aurora	17	elasticsearch	13
redis	18	Google Spanner	17	PostgreSQL	13
cassandra	17	snowflake	17	Amazon Aurora	12
elasticsearch	12	PostgreSQL	17	SQLite	11
HIVE	11	OceanBase	16	MySQL	11
Scuba	10	amazon REDSHIFT	15	splunk>	9
MySQL	10	elasticsearch	15	RocksDB	9

amazon DynamoDB

HISTORY

Amazon publishes a paper in 2007 on the original Dynamo system.

→ Eventually consistency key/value store
→ Shared-nothing architecture
→ Non-SQL API, no joins, no transactions
→ Partitions based on consistent hashing

Amazon makes DynamoDB available to customers on AWS in 2012.

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

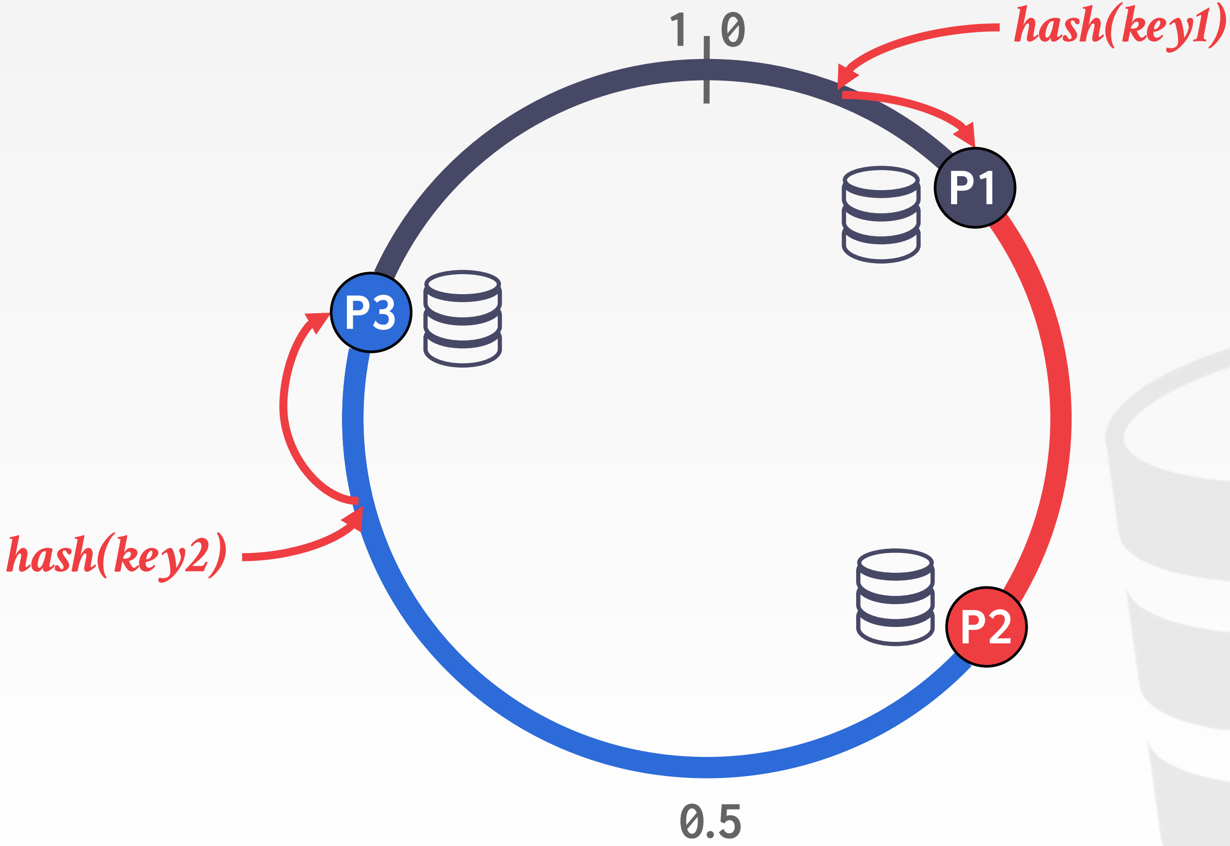
To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

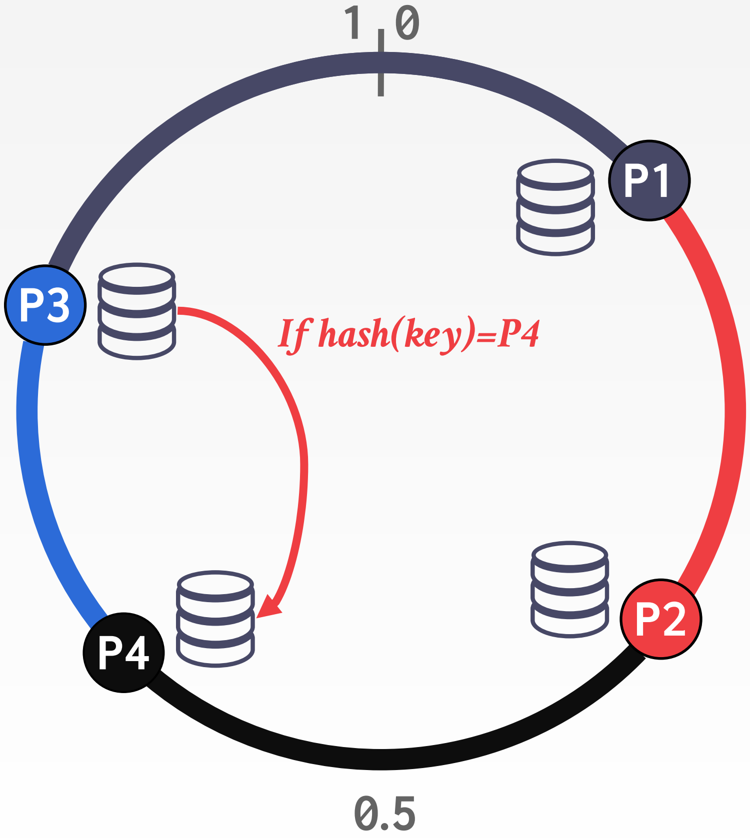
Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

205

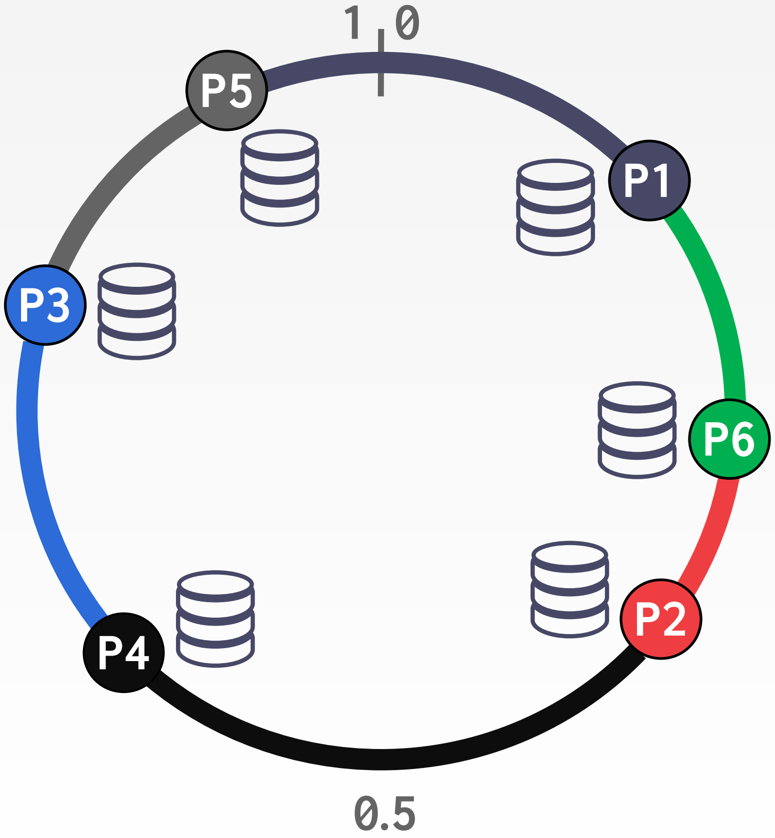
CONSISTENT HASHING



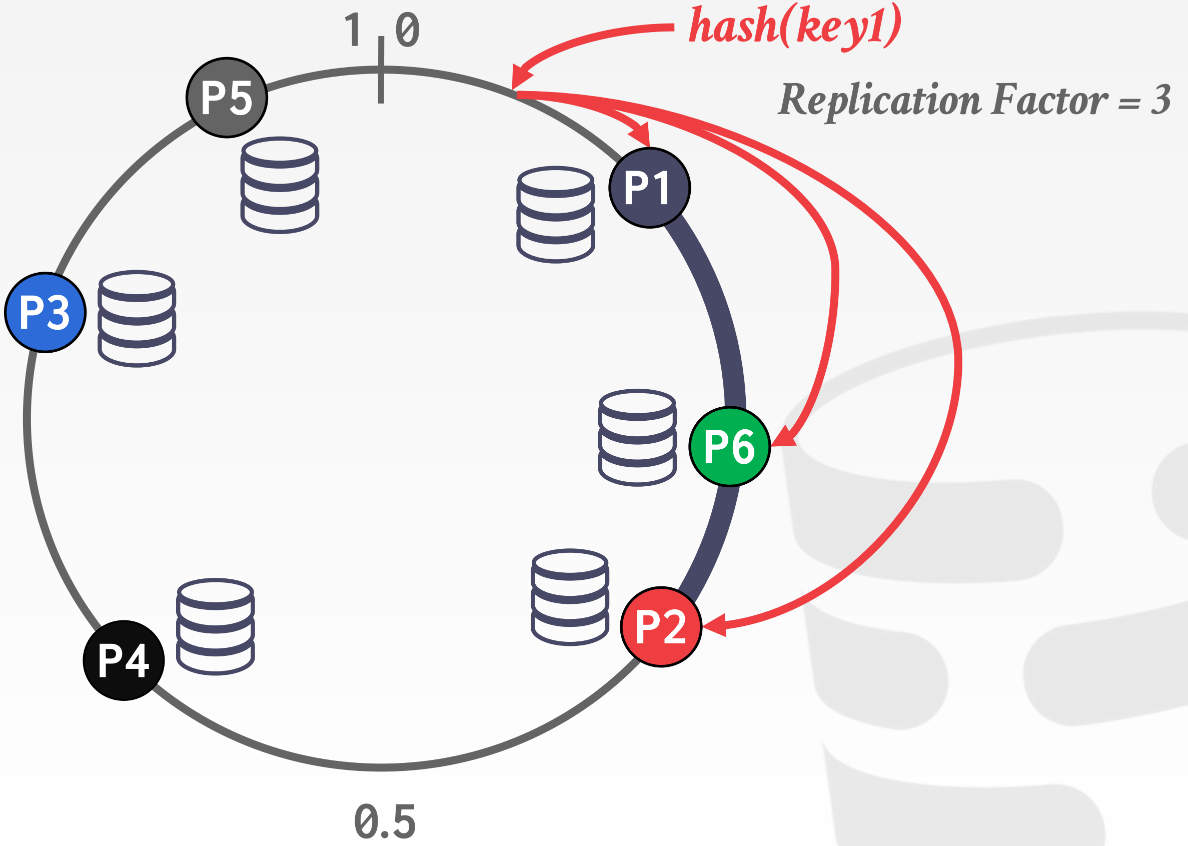
CONSISTENT HASHING



CONSISTENT HASHING



CONSISTENT HASHING



DYNAMODB DATA MODEL

DynamoDB supports a subset of the relational model:

→ A table definition includes the set of attributes that the table's records must contain.
→ You cannot specify constraints (integrity, foreign key).

Tables support two types of primary keys:

→ Single Partition Key
→ Composite Partition Key + Sort Key

DYNAMODB SECONDARY INDEXES

Local Secondary Index

→ Use the partition key to initially route the request to a node.
→ Each node maintains a local B+Tree that only contains keys for the records stored at that node.

Global Secondary Index

→ Uses a partition key + sort key that is different than the table's primary key.
→ Not guaranteed to be consistent with the table.

DYNAMODB API

Access the database using API calls.
→ Application must perform additional operations client-side if they need functionality beyond provided API.

Modification API calls support application-specified conditionals to deal with eventual consistency issues.

Read Data

```
GetItem
BatchGetItem
Scan
Query
```

Modify Data

```
PutItem
BatchWriteItem
UpdateItem
DeleteItem
```

DYNAMODB TRANSACTIONS

In 2018, Amazon announced support for client-side transaction support in DynamoDB.
→ Centralized Middleware Coordinator

Called "single-shot" transactions because you need to know your read/write set before the transaction starts.

```
TransactionGetItems(
  Get(table:T1, key:k1),
  Get(table:T2, key:k2),
  Get(table:T3, key:k3)
)
```

```
TransactionWriteItems(
  Put(table:T1, key:k1, value:v1),
  Delete(table:T2, key:k2),
  Update(table:T3, key:k3),
  Check(table:T3, key:k3, value:<100)
)
```

Source: Doug Terry

DYNAMODB TRANSACTIONS

Move $100 from Andy' bank account to his promotor's account.

```
BEGIN
A = A - 100
B = B + 100
COMMIT
```

```
A = Get(person:"Andy")
B = Get(person:"Bookie")
TransactionWriteItems(
  Check(person:"Andy", balance:A),
  Check(person:"Bookie", balance:B),
  Put(person:"Andy", balance:A-100),
  Put(person:"Bookie", balance:B+100)
)
```

Source: Doug Terry

cassandra

HISTORY

After Amazon published the Dynamo paper in 2007, people at Facebook start writing a clone called Cassandra in 2008 for their message service.

→ Decided to not use the DBMS and instead released the source code.

Picked up by organizations outside of Facebook and then became an Apache project in 2009.

APACHE CASSANDRA

Borrows a lot of ideas from other systems:
→ Eventual Consistency
→ Shared-Nothing
→ Consistent Hashing (Amazon Dynamo)
→ Column-Family Data Model (Google BigTable)
→ Log-structured Merge Trees

Originally one of the main proponents of the NoSQL movement but now pushing **CQL**.

LSM STORAGE MODEL

The log is the database.
→ DBMS reads log to reconstruct the record for a read.

MemTable: In-memory cache

SSTables:
→ Read-only portions of the log.
→ Use indexes + Bloom filters to speed up reads
→ See the CMU-DB **RocksDB** talk (2015)
http://cmudb.io/lectures2015-rocksdb

mongoDB

MONGODB

Distributed **document** DBMS started in 2007.
→ Document → Tuple
→ Collection → Table/Relation

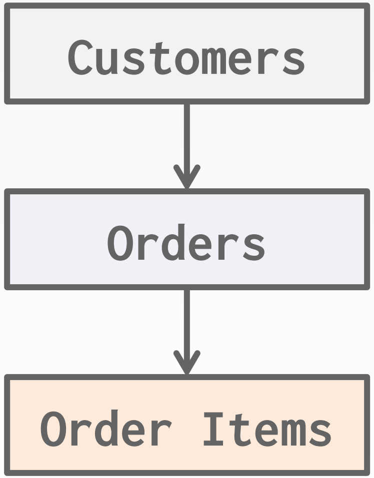
Open-source (Server Side Public License)

Centralized shared-nothing architecture.

Concurrency Control:
→ OCC with multi-granular locking

PHYSICAL DENORMALIZATION

A **CUSTOMER** has one or more **ORDER** records. Each **ORDER** record has one or more **ORDER_ITEM** records.



$\rightarrow R_1(\texttt{custId,name,…})$

$\bowtie$

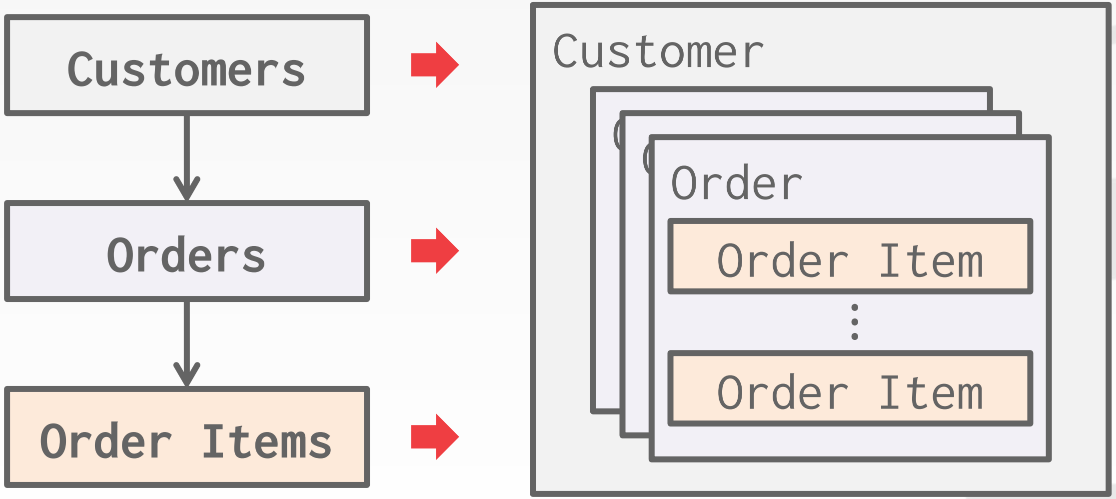
$\rightarrow R_2(\texttt{orderId,custId,…})$

$\bowtie$

$\rightarrow R_3(\texttt{itemId,orderId,…})$

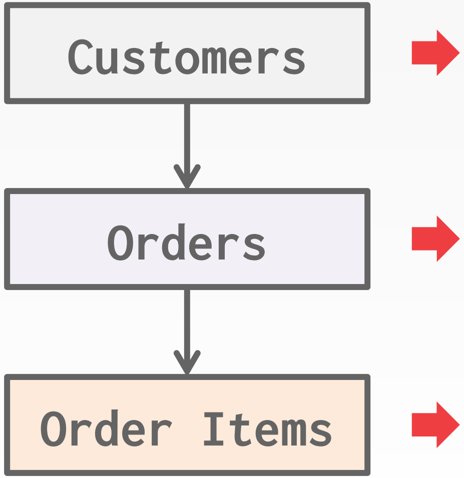
PHYSICAL DENORMALIZATION

A **CUSTOMER** has one or more **ORDER** records. Each **ORDER** record has one or more **ORDER_ITEM** records.



PHYSICAL DENORMALIZATION

A **CUSTOMER** has one or more **ORDER** records. Each **ORDER** record has one or more **ORDER_ITEM** records.




```
{
  "custId": 1234,
  "custName": "Andy",
  "orders": [
    { "orderId": 9999,
      "orderItems": [
        { "itemId": "XXXX",
          "price":  19.99 },
        { "itemId": "YYYY",
          "price":  29.99 },
      ] }
  ]
}
```

QUERY EXECUTION

JSON-only query API

No cost-based query planner / optimizer.
→ Heuristic-based + "random walk" optimization.

JavaScript UDFs (not encouraged).

Supports server-side joins (only left-outer?).

Multi-document transactions.

DISTRIBUTED ARCHITECTURE

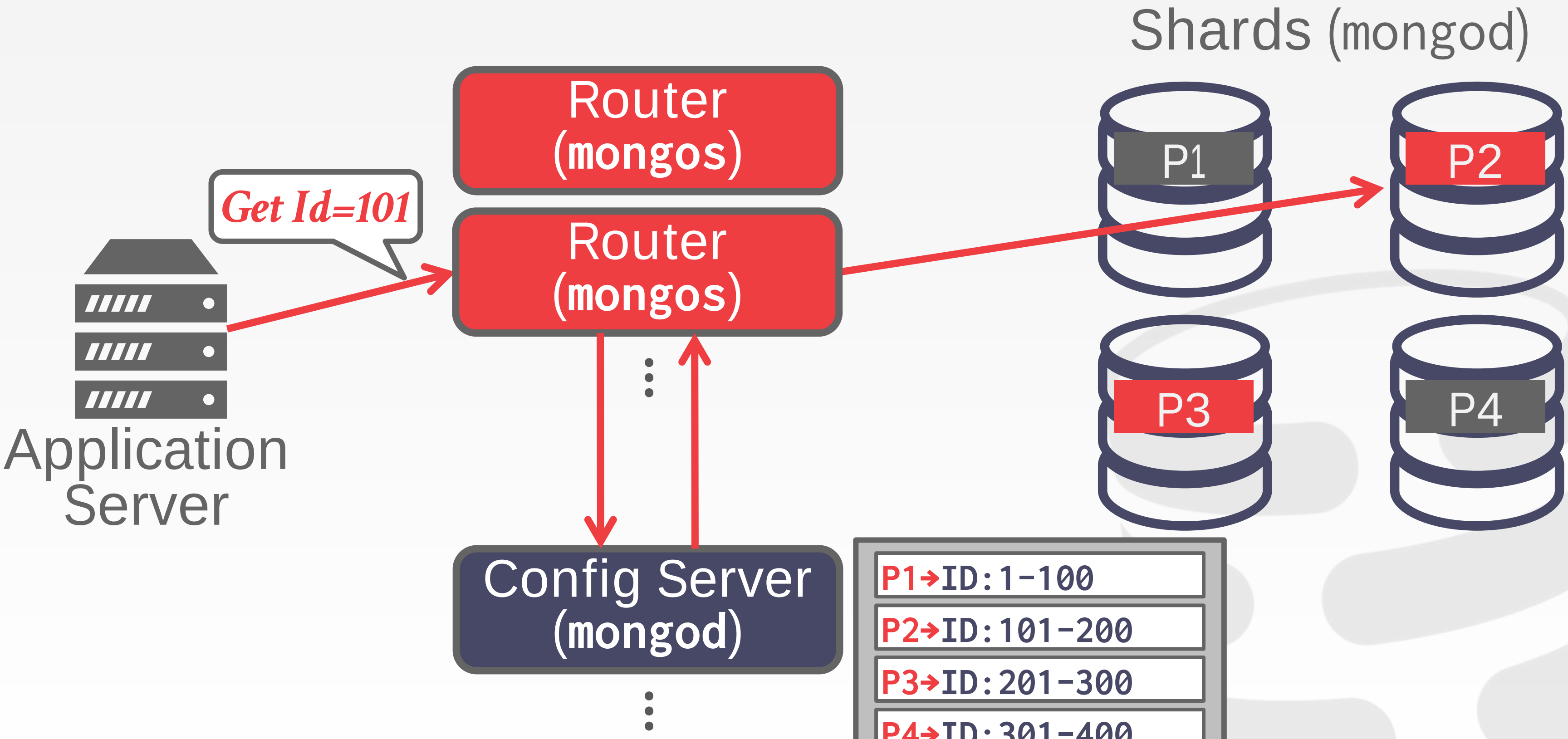
Heterogeneous distributed components.
→ Shared nothing architecture
→ Centralized query router.

Master-slave replication.

Auto-sharding:
→ Define 'partitioning' attributes for each collection (hash or range).
→ When a shard gets too big, the DBMS automatically splits the shard and rebalances.

mongoDB

MONGODB CLUSTER ARCHITECTURE



STORAGE ARCHITECTURE

Originally used **mmap** storage manager
→ No buffer pool.
→ Let the OS decide when to flush pages.
→ Single lock per database.

MongoDB v3 supports pluggable storage backends
→ **WiredTiger** from BerkeleyDB alumni.
http://cmudb.io/lectures2015-wiredtiger
→ **RocksDB** from Facebook ("MongoRocks")
http://cmudb.io/lectures2015-rocksdb

mangoDB

MANGODB

Single-node satirical implementation of MongoDB written in Python.

→ Only supports MongoDB wire protocol v2

All data is written to **`/dev/null`**

https://github.com/dcramer/mangodb

ANDY'S CONCLUDING REMARKS

Databases are awesome.
→ They cover all facets of computer science.
→ We have barely scratched the surface…

Going forth, you should now have a good understanding how these systems work.

This will allow you to make informed decisions throughout your entire career.
→ Avoid premature optimizations.