Carnegie Mellon University

Final Review + Systems Potpourri

Lecture #26

Database Systems
15-445/15-645
Fall 2018

AP
Andy Pavlo
Computer Science
Carnegie Mellon Univ.

ADMINISTRIVIA

Project #4: Monday Dec 10th @ 11:59pm

Extra Credit: Wednesday Dec 12th @11:59pm

Final Exam: Sunday Dec 16th @ 8:30am

FINAL EXAM

Who: You

What: http://cmudb.io/f18-final

When: Sunday Dec 16th @ 8:30am

Where: GHC 4401

Why: https://youtu.be/6yOH_FjeSAQ

FINAL EXAM

What to bring:

→ CMU ID
→ Calculator
→ Two pages of handwritten notes (double-sided)

Optional:

→ Spare change of clothes

What not to bring:

→ Your roommate

COURSE EVALS

Your feedback is strongly needed:
→ https://cmu.smartevals.com

Things that we want feedback on:
→ Homework Assignments
→ Projects
→ Reading Materials
→ Lectures

OFFICE HOURS

Andy:

→ Friday Dec. 14th @ 3:00pm-4:00pm

STUFF BEFORE MID-TERM

SQL
Buffer Pool Management
Hash Tables
B+Trees
Storage Models

PARALLEL EXECUTION

Inter-Query Parallelism

Intra-Query Parallelism

Inter-Operator Parallelism

Intra-Operator Parallelism

EMBEDDED LOGIC

User-defined Functions

Stored Procedures

Focus on advantages vs. disadvantages

TRANSACTIONS

ACID

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

View Serializability

Recoverable Schedules

Isolation Levels / Anomalies

TRANSACTIONS

Two-Phase Locking
→ Strict vs. Non-Strict
→ 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

2015

MongoDB	32
Google Spanner/F1	22
LinkedIn Espresso	16
Apache Cassandra	16
Facebook Scuba	16
Apache Hbase	14
VoltDB	10
Redis	10
Vertica	5
Cloudera Impala	5

2016

MongoDB	33
Google Spanner/F1	22
Apache Cassandra	19
Facebook Scuba	17
Redis	16
Apache Hbase	15
CockroachDB	12
LinkedIn Espresso	11
Cloudera Impala	8
Peloton	7

2017

Google Spanner/F1	15
MongoDB	14
CockroachDB	10
Apache Hbase	9
Peloton	8
Facebook Scuba	6
Cloudera Impala	6
Apache Hive	6
Apache Cassandra	5
LinkedIn Espresso	5

2018

CockroachDB	26
Google Spanner/F1	25
MongoDB	24
Amazon Aurora	18
Redis	18
Apache Cassandra	17
ElasticSearch	12
Apache Hive	11
Facebook Scuba	10
MySQL	10

Cockroach LABS

COCKROACHDB

Started in 2015 by ex-Google employees.

Open-source (Apache Licensed)

Decentralized shared-nothing architecture.

Log-structured on-disk storage (RocksDB)

Concurrency Control:
- → MVCC + OCC
- → Serializable isolation only

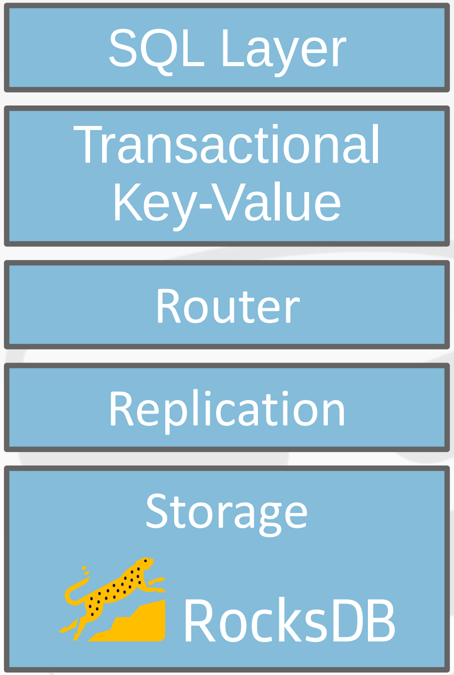
DISTRIBUTED ARCHITECTURE

Multi-layer architecture on top of a replicated key-value store.

→ All tables and indexes are store in a giant sorted map in the k/v store.

Uses RocksDB as the storage manager at each node.

Raft protocol (variant of Paxos) for replication and consensus.



CONCURRENCY CONTROL

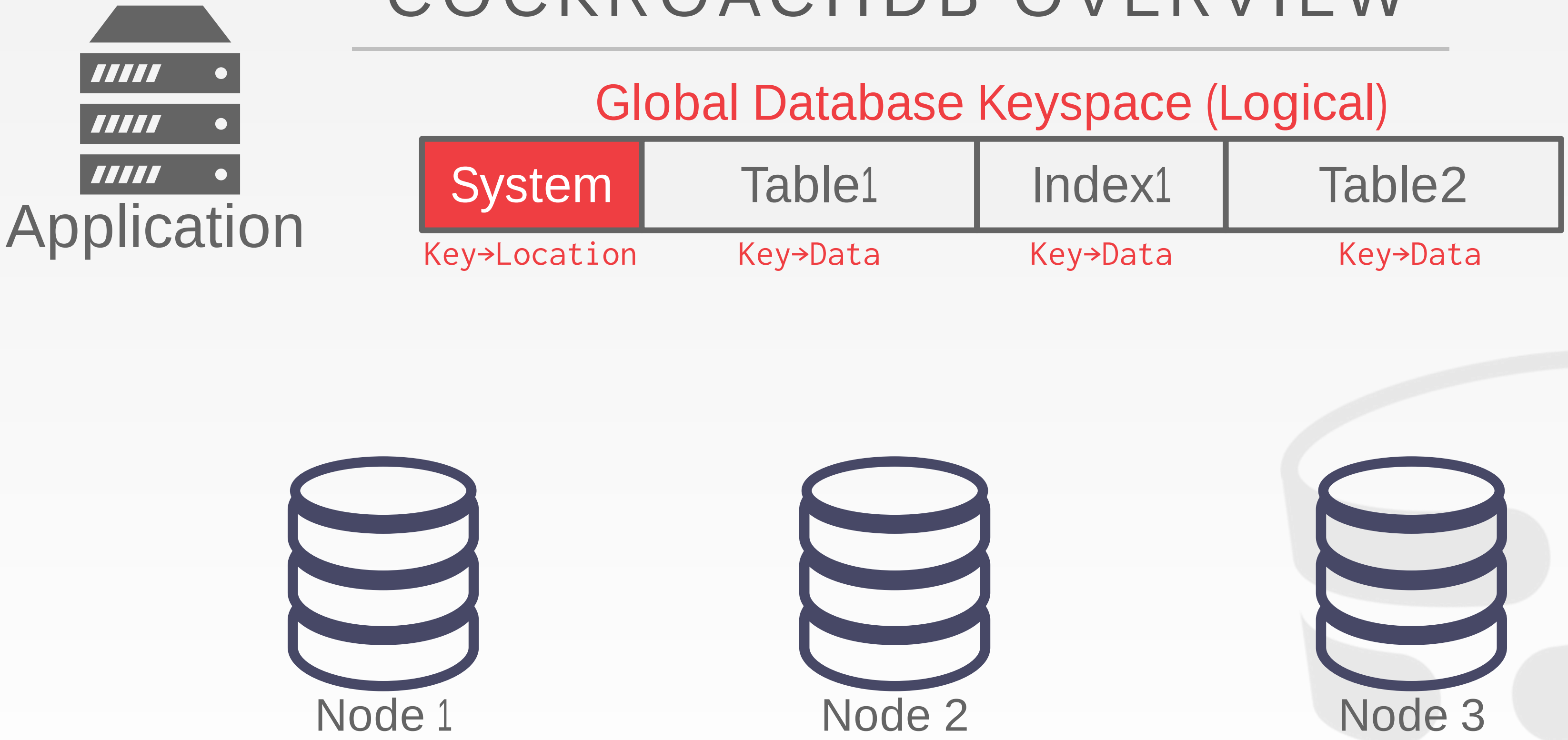
DBMS uses hybrid clocks (physical + logical) to order transactions globally.

→ Synchronized wall clock with local counter.

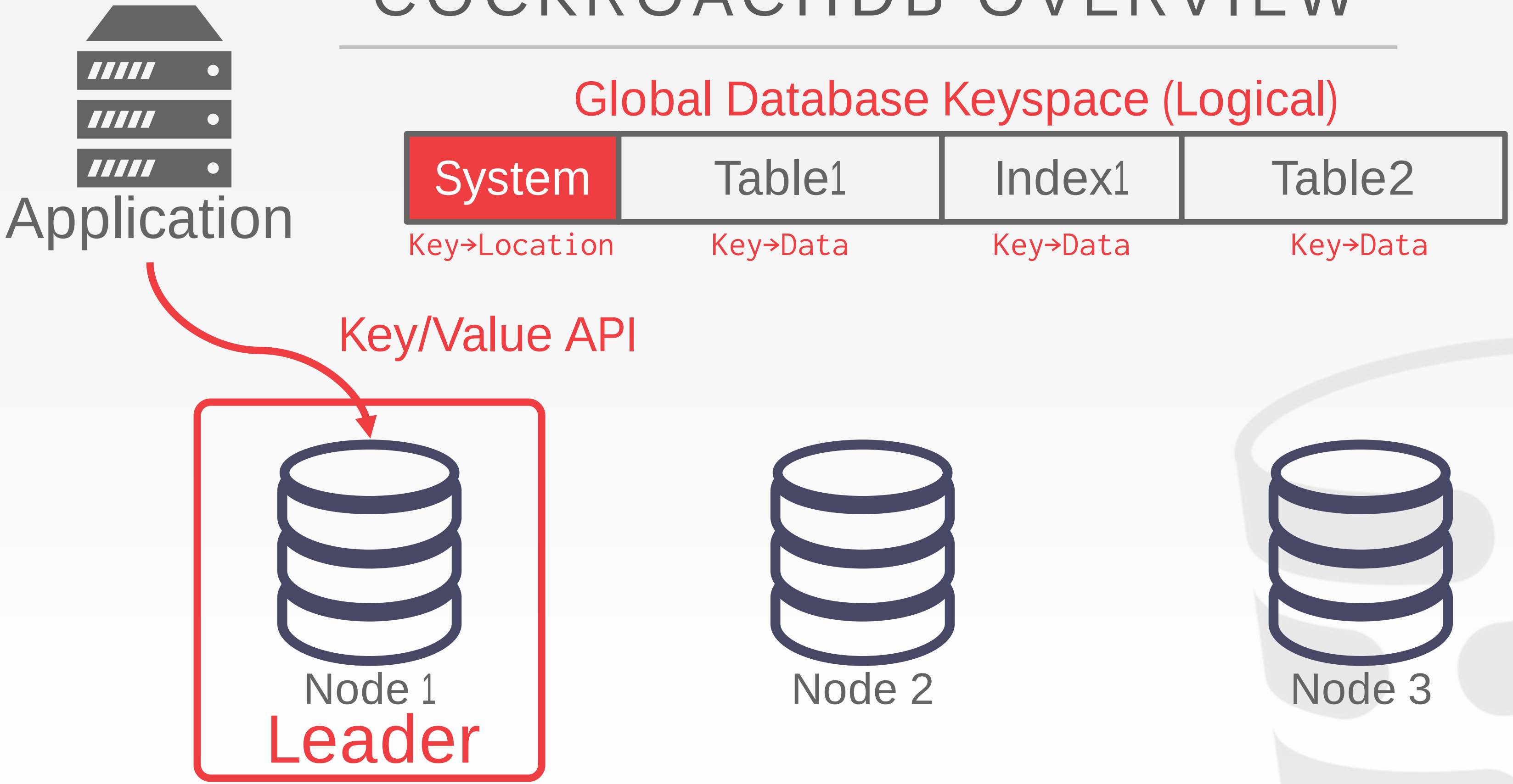
Txns stage writes as "intents" and then checks for conflicts on commit.

All meta-data about txns state resides in the key-value store.

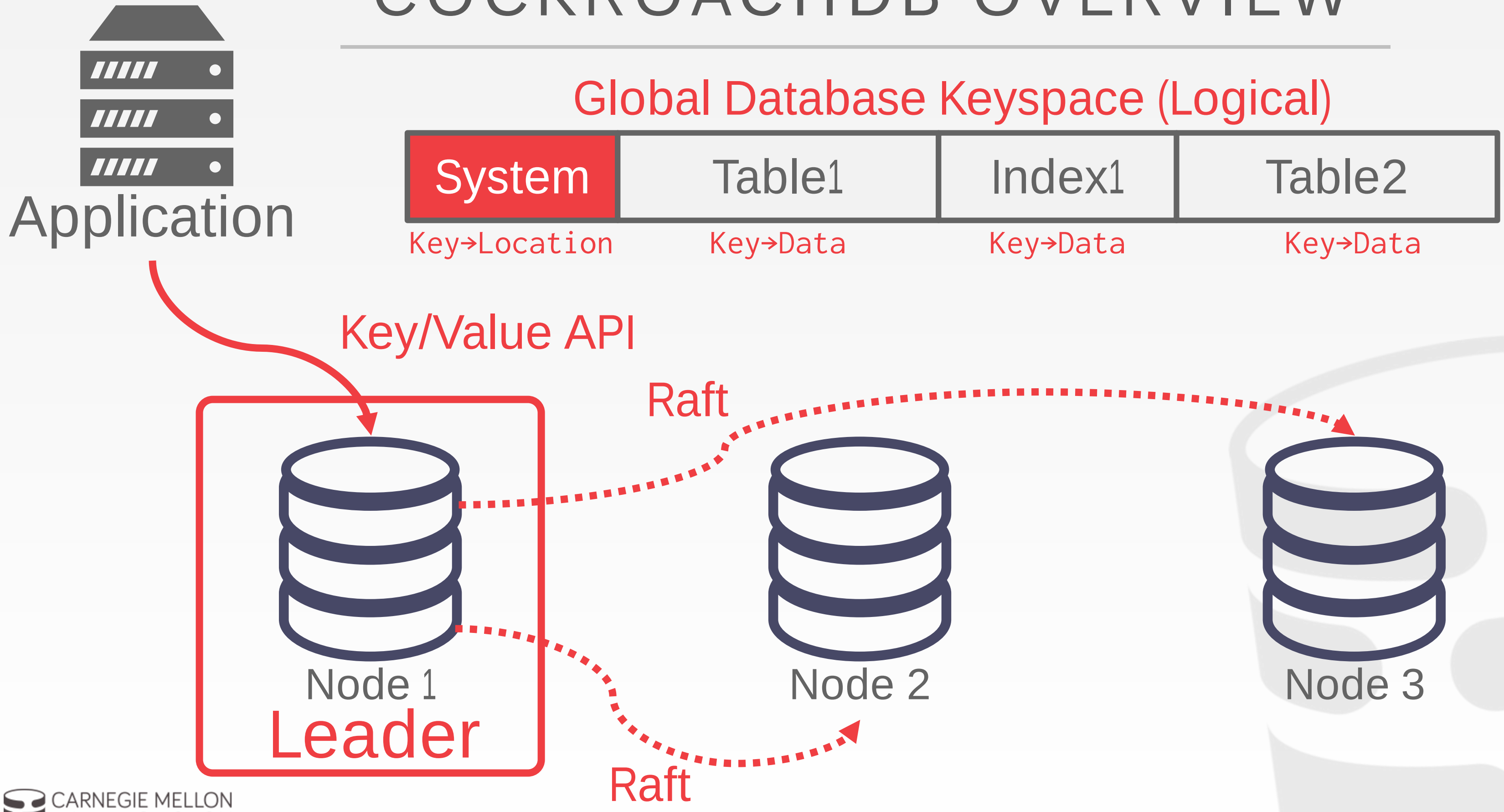
COCKROACHDB OVERVIEW



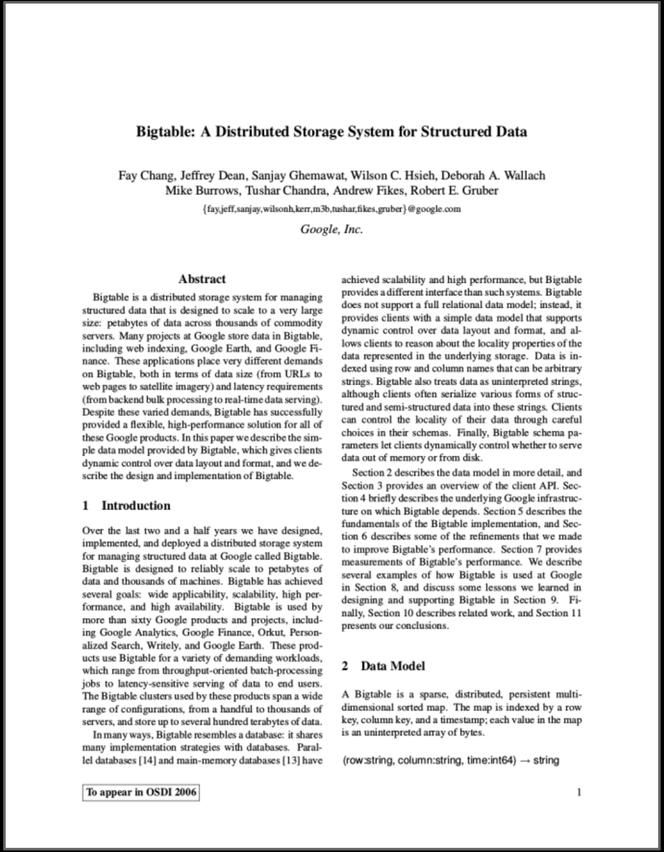
COCKROACHDB OVERVIEW



COCKROACHDB OVERVIEW



Google Spanner



Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

To appear in OSDI 2006

2006

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google, Inc.

{jasonbaker,chrisbond,jcorbett,jfurman,akhorlin,jimlarson,jm,yaweili,alloyd,vadimy}@google.com

ABSTRACT

Megastore is a storage system developed to meet the requirements of today's interactive online services. Megastore blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS in a novel way, and provides both strong consistency guarantees and high availability. We provide fully serializable ACID semantics within fine-grained partitions of data. This partitioning allows us to synchronously replicate each write across a wide area network with reasonable latency and support seamless failover between datacenters. This paper describes Megastore's semantics and replication algorithm. It also describes our experience supporting a wide range of Google production services built with Megastore.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases; H.2.4 [**Database Management**]: Systems—*concurrency, distributed databases*

General Terms

Algorithms, Design, Performance, Reliability

Keywords

Large databases, Distributed transactions, Bigtable, Paxos

1. INTRODUCTION

Interactive online services are forcing the storage community to meet new demands as desktop applications migrate to the cloud. Services like email, collaborative documents, and social networking have been growing exponentially and are testing the limits of existing infrastructure. Meeting these services' storage demands is challenging due to a number of conflicting requirements.

First, the Internet brings a huge audience of potential users, so the applications must be *highly scalable*. A service can be built rapidly using MySQL [10] as its datastore, but scaling the service to millions of users requires a complete redesign of its storage infrastructure. Second, services must compete for users. This requires *rapid development* of features and fast time-to-market. Third, the service must be responsive; hence, the storage system must have *low latency*. Fourth, the service should provide the user with a *consistent view of the data*—the result of an update should be visible immediately and durably. Seeing edits to a cloud-hosted spreadsheet vanish, however briefly, is a poor user experience. Finally, users have come to expect Internet services to be up 24/7, so the service must be *highly available*. The service must be resilient to many kinds of faults ranging from the failure of individual disks, machines, or routers all the way up to large-scale outages affecting entire datacenters.

These requirements are in conflict. Relational databases provide a rich set of features for easily building applications, but they are difficult to scale to hundreds of millions of users. NoSQL datastores such as Google's Bigtable [15], Apache Hadoop's HBase [1], or Facebook's Cassandra [6] are highly scalable, but their limited API and loose consistency models complicate application development. Replicating data across distant datacenters while providing low latency is challenging, as is guaranteeing a consistent view of replicated data, especially during faults.

Megastore is a storage system developed to meet the storage requirements of today's interactive online services. It is novel in that it blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS. It uses synchronous replication to achieve high availability and a consistent view of the data. In brief, it provides fully serializable ACID semantics over distant replicas with low enough latencies to support interactive applications.

We accomplish this by taking a middle ground in the RDBMS vs. NoSQL design space: we partition the datastore and replicate each partition separately, providing full ACID semantics within partitions, but only limited consistency guarantees across them. We provide traditional database features, such as secondary indexes, but only those features that can scale within user-tolerable latency limits, and only with the semantics that our partitioning scheme can support. We contend that the data for most Internet services can be suitably partitioned (e.g., by user) to make this approach viable, and that a small, but not spartan, set of features can substantially ease the burden of developing cloud applications.

Contrary to conventional wisdom [24, 28], we were able to use Paxos [27] to build a highly available system that pro-

5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

2011

GOOGLE SPANNER

Google's geo-replicated DBMS (>2011)

Schematized, semi-relational data model.

Decentralized shared-disk architecture.

Log-structured on-disk storage.

Concurrency Control:

→ Strict 2PL + MVCC + Multi-Paxos + 2PC
→ **Externally consistent** global write-transactions with synchronous replication.
→ Lock-free read-only transactions.

PHYSICAL DENORMALIZATION

```
CREATE TABLE users {
  uid INT NOT NULL,
  email VARCHAR,
  PRIMARY KEY (uid)
};
CREATE TABLE albums {
  uid INT NOT NULL,
  aid INT NOT NULL,
  name VARCHAR,
  PRIMARY KEY (uid, aid)
} INTERLEAVE IN PARENT users
  ON DELETE CASCADE;
```

PHYSICAL DENORMALIZATION

```
CREATE TABLE users {
  uid INT NOT NULL,
  email VARCHAR,
  PRIMARY KEY (uid)
};
CREATE TABLE albums {
  uid INT NOT NULL,
  aid INT NOT NULL,
  name VARCHAR,
  PRIMARY KEY (uid, aid)
} INTERLEAVE IN PARENT users
  ON DELETE CASCADE;
```

Physical Storage

users(1001)
↳albums(1001, 9990)
↳albums(1001, 9991)

users(1002)
↳albums(1002, 6631)
↳albums(1002, 6634)

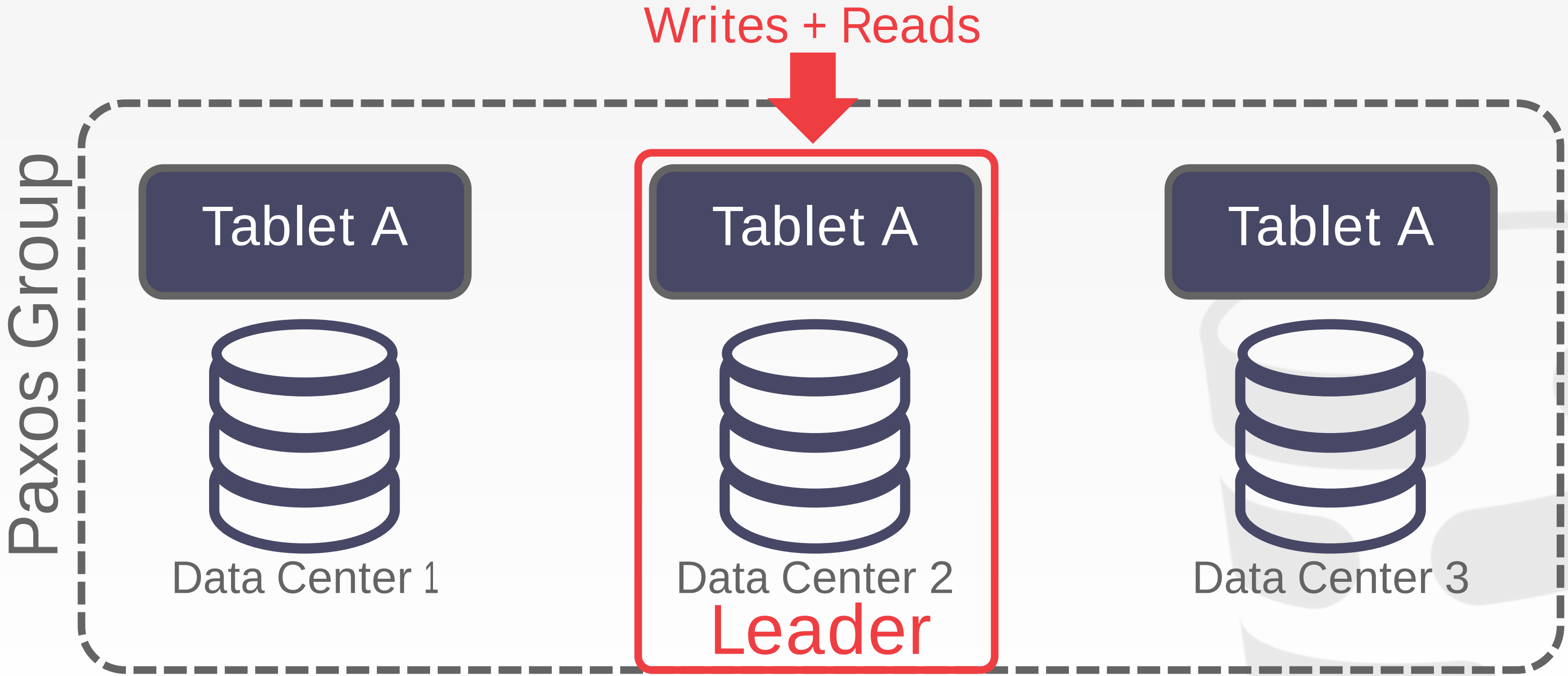
CONCURRENCY CONTROL

MVCC + Strict 2PL with Wound-Wait Deadlock Prevention

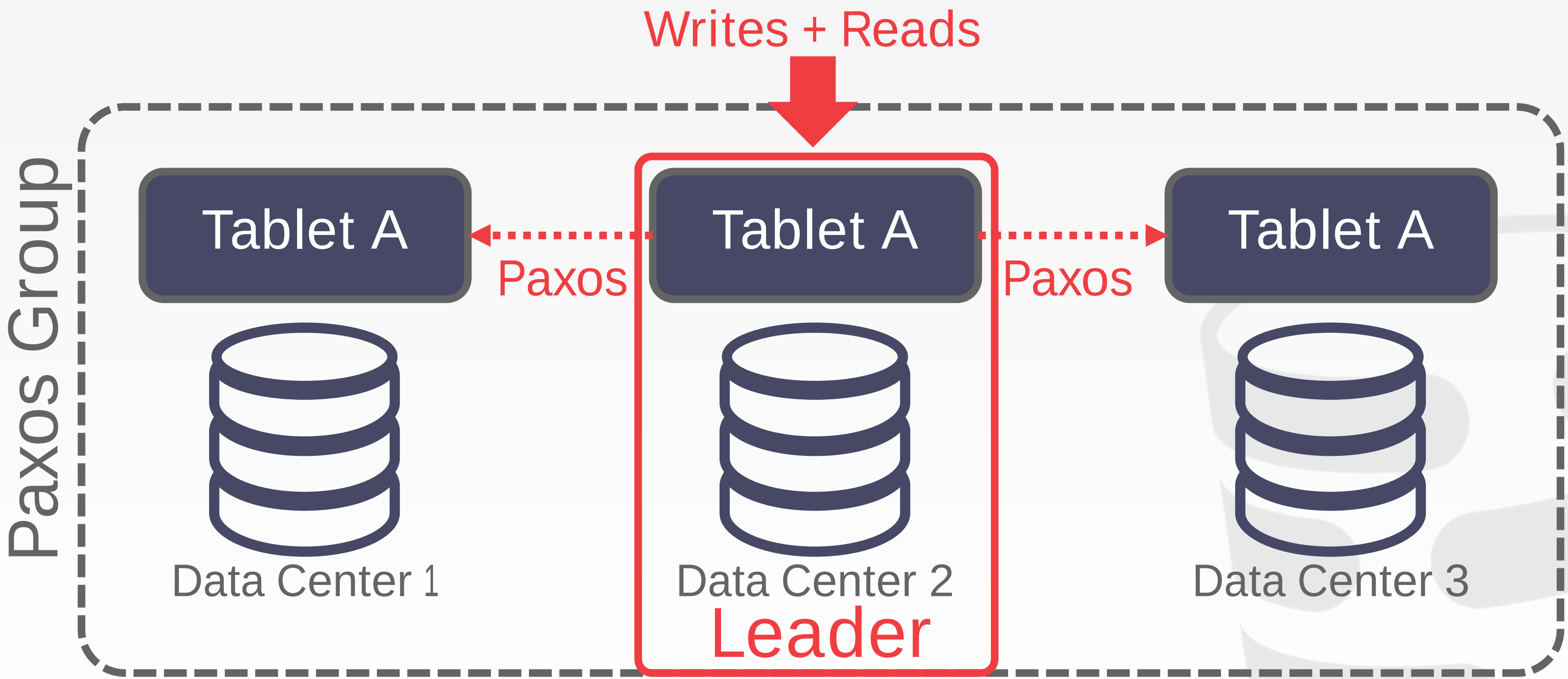
Ensures ordering through globally unique timestamps generated from atomic clocks and GPS devices.

Database is broken up into tablets:
→ Use Paxos to elect leader in tablet group.
→ Use 2PC for txns that span tablets.

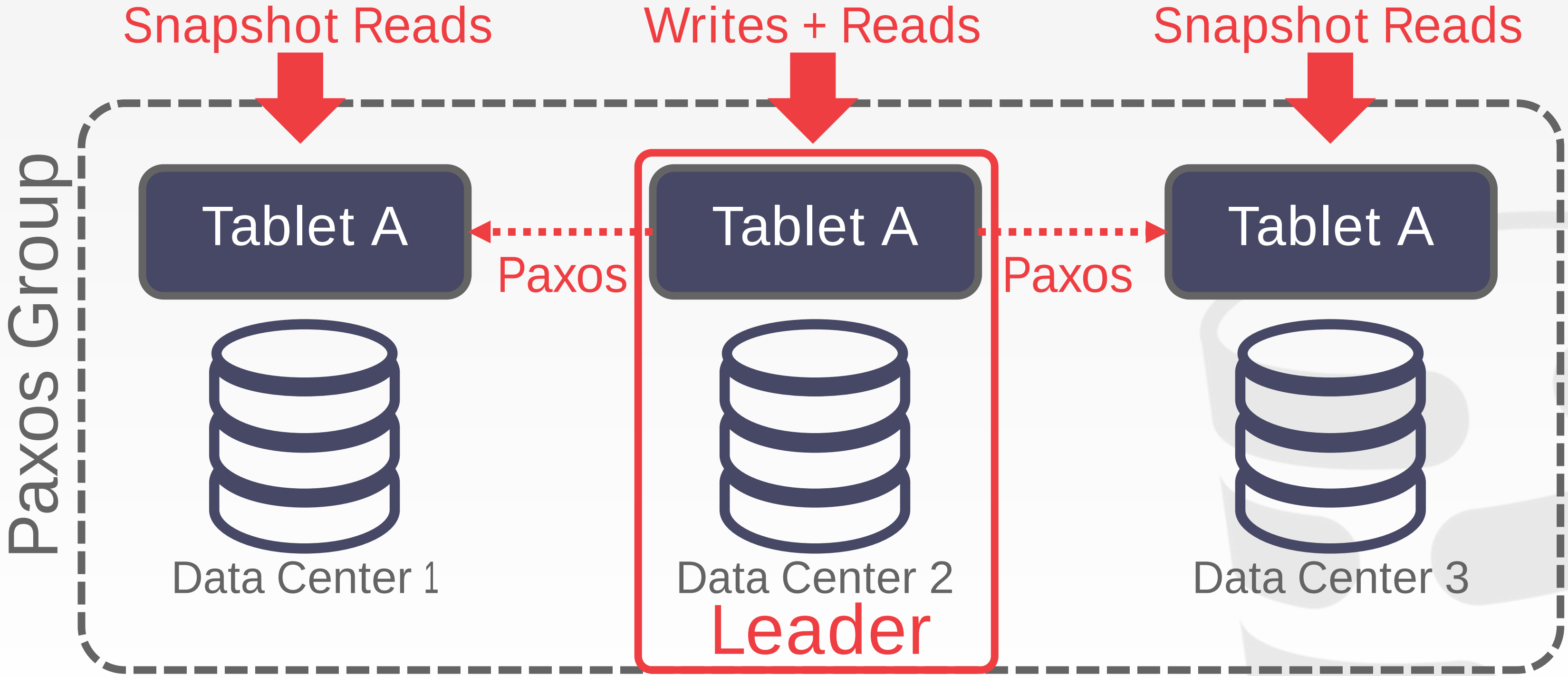
SPANNER TABLETS



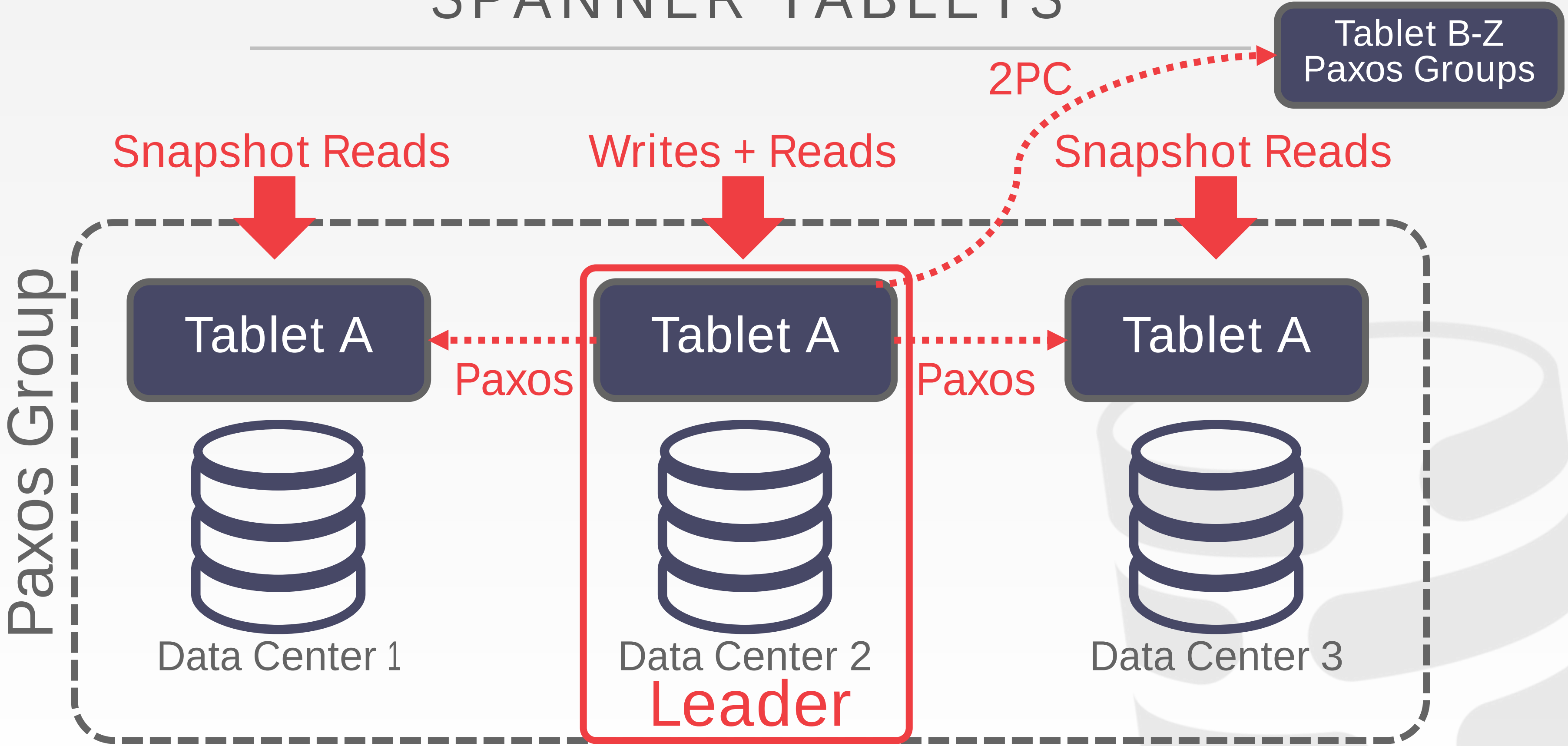
SPANNER TABLETS



SPANNER TABLETS



SPANNER TABLETS



TRANSACTION ORDERING

Spanner orders transactions based on physical "wall-clock" time.

→ This is necessary to guarantee linearizability.

→ If T_1 finishes before T_2, then T_2 should see the result of T_1.

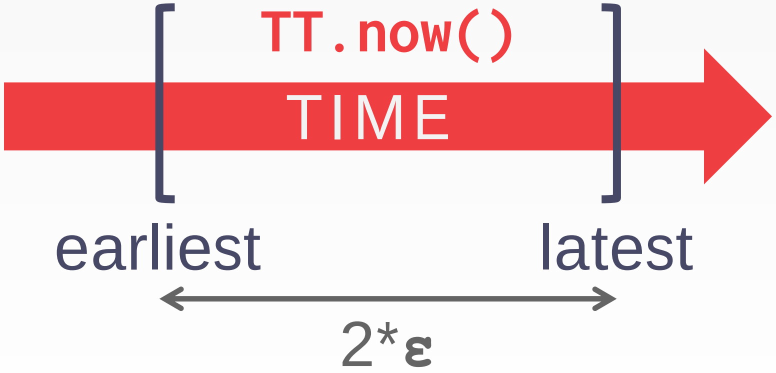
Each Paxos group decides in what order transactions should be committed according to the timestamps.

→ If T_1 commits at $time_1$ and T_2 starts at $time_2 > time_1$, then T_1's timestamp should be less than T_2's.

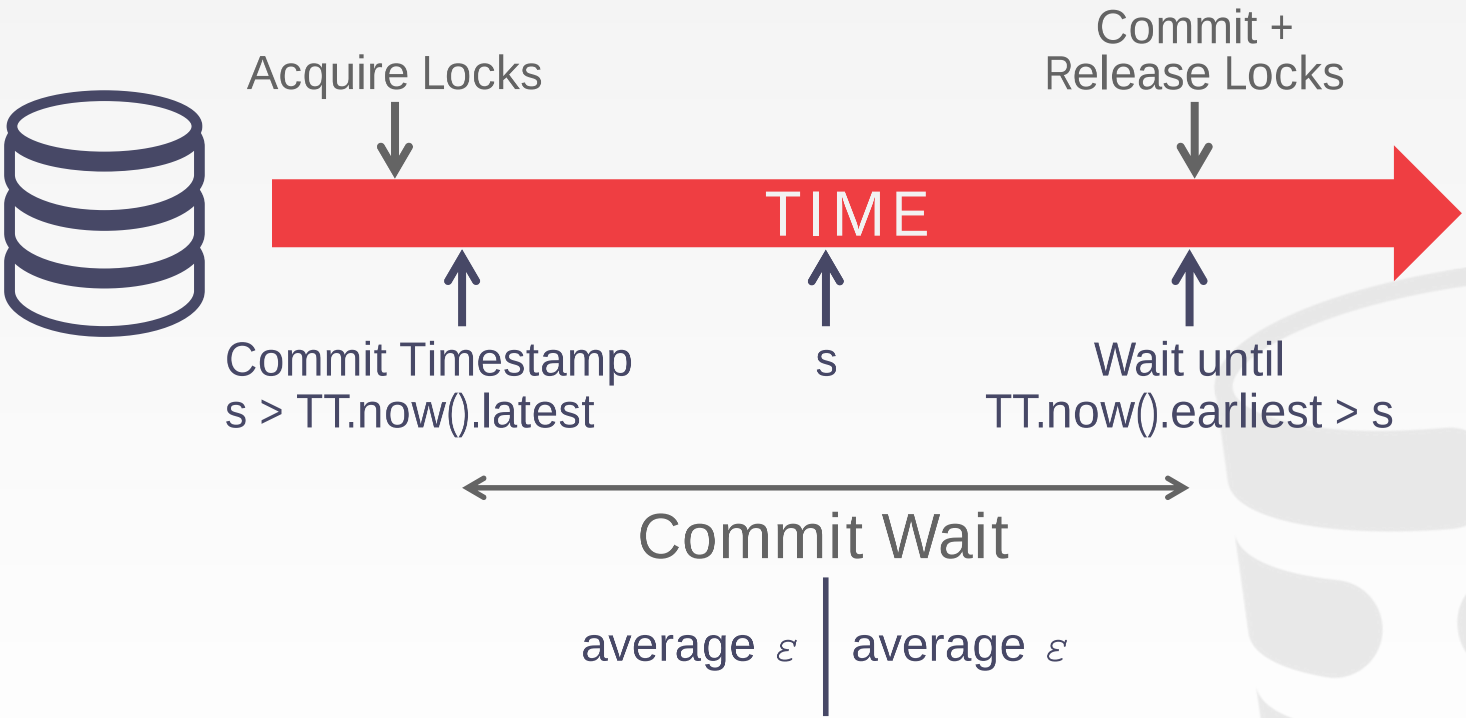
SPANNER TRUETIME

The DBMS maintains a global wall-clock time across all data centers with bounded uncertainty.

Timestamps are intervals, not single values



SPANNER TRUETIME



GOOGLE F1 (2013)

OCC engine built on top of Spanner.

→ In the read phase, F1 returns the last modified timestamp with each row. No locks.

→ The timestamp for a row is stored in a hidden lock column. The client library returns these timestamps to the F1 server.

→ If the timestamps differ from the current timestamps at the time of commit the transaction is aborted.

F1: A Distributed SQL Database That Scales

Jeff Shute, Chad Whipkey, David Menestrina, Radek Vingralek, Eric Rollins, Stephan Ellner, Traian Stancescu, Bart Samwel, Mircea Oancea, John Cieslewicz, Himani Apte, Ben Handy, Kyle Littlefield, Ian Rae*

Google, Inc.
*University of Wisconsin-Madison

ABSTRACT

F1 is a distributed relational database system built at Google to support the AdWords business. F1 is a hybrid database that combines high availability, the scalability of NoSQL systems like Bigtable, and the consistency and usability of traditional SQL databases. F1 is built on Spanner, which provides synchronous cross-datacenter replication and strong consistency. Synchronous replication implies higher commit latency, but we mitigate that latency by using a hierarchical schema model with structured data types and through smart application design. F1 also includes a fully functional distributed SQL query engine and automatic change tracking and publishing.

1. INTRODUCTION

F1[1] is a fault-tolerant globally-distributed OLTP and OLAP database built at Google as the new storage system for Google's AdWords system. It was designed to replace a sharded MySQL implementation that was not able to meet our growing scalability and reliability requirements.

The key goals of F1's design are:

1. **Scalability**: The system must be able to scale up, trivially and transparently, just by adding resources. Our sharded database based on MySQL was hard to scale up, and even more difficult to rebalance. Our users needed complex queries and joins, which meant they had to carefully shard their data, and resharding data without breaking applications was challenging.
2. **Availability**: The system must never go down for any reason – datacenter outages, planned maintenance, schema changes, etc. The system stores data for Google's core business. Any downtime has a significant revenue impact.
3. **Consistency**: The system must provide ACID transactions, and must always present applications with consistent and correct data.

   Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains.
4. **Usability**: The system must provide full SQL query support and other functionality users expect from a SQL database. Features like indexes and ad hoc query are not just nice to have, but absolute requirements for our business.

Recent publications have suggested that these design goals are mutually exclusive [5, 11, 23]. A key contribution of this paper is to show how we achieved all of these goals in F1's design, and where we made trade-offs and sacrifices. The name F1 comes from genetics, where a *Filial 1 hybrid* is the first generation offspring resulting from a cross mating of distinctly different parental types. The F1 database system is indeed such a hybrid, combining the best aspects of traditional relational databases and scalable NoSQL systems like Bigtable [6].

F1 is built on top of Spanner [7], which provides extremely scalable data storage, synchronous replication, and strong consistency and ordering properties. F1 inherits those features from Spanner and adds several more:

- Distributed SQL queries, including joining data from external data sources
- Transactionally consistent secondary indexes
- Asynchronous schema changes including database reorganizations
- Optimistic transactions
- Automatic change history recording and publishing

Our design choices in F1 result in higher latency for typical reads and writes. We have developed techniques to hide that increased latency, and we found that user-facing transactions can be made to perform as well as in our previous MySQL system:

- An F1 schema makes data clustering explicit, using tables with hierarchical relationships and columns with structured data types. This clustering improves data locality and reduces the number and cost of RPCs required to read remote data.

[1] Previously described briefly in [22].

Proceedings of the VLDB Endowment, Vol. 6, No. 11

GOOGLE CLOUD SPANNER (2017)

Spanner Database-as-a-Service.

AFAIK, it is based on Spanner SQL not F1.

Spanner: Becoming a SQL System

David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, Dale Woodford

Google, Inc.

ABSTRACT

Spanner is a globally-distributed data management system that backs hundreds of mission-critical services at Google. Spanner is built on ideas from both the systems and database communities. The first Spanner paper published at OSDI'12 focused on the systems aspects such as scalability, automatic sharding, fault tolerance, consistent replication, external consistency, and wide-area distribution. This paper highlights the database DNA of Spanner. We describe distributed query execution in the presence of resharding, query restarts upon transient failures, range extraction that drives query routing and index seeks, and the improved blockwise-columnar storage format. We touch upon migrating Spanner to the common SQL dialect shared with other systems at Google.

1. INTRODUCTION

Google's Spanner [5] started out as a key-value store offering multi-row transactions, external consistency, and transparent failover across datacenters. Over the past 7 years it has evolved into a relational database system. In that time we have added a strongly-typed schema system and a SQL query processor, among other features. Initially, some of these database features were "bolted on" – the first version of our query system used high-level APIs almost like an external application, and its design did not leverage many of the unique features of the Spanner storage architecture. However, as we have developed the system, the desire to make it behave more like a traditional database has forced the system to evolve. In particular,

- The architecture of the distributed storage stack has driven fundamental changes in our query compilation and execution, and
- The demands of the query processor have driven fundamental changes in the way we store and manage data.

These changes have allowed us to preserve the massive scalability of Spanner, while offering customers a powerful platform for database applications. We have previously described the distributed architecture and data and concurrency model of Spanner [5]. In this paper, we focus on the "database system" aspects of Spanner, in particular how query execution has evolved and forced the rest of Spanner to evolve. Most of these changes have occurred since [5] was written, and in many ways today's Spanner is very different from what was described there.

A prime motivation for this evolution towards a more "database-like" system was driven by the experiences of Google developers trying to build on previous "key-value" storage systems. The prototypical example of such a key-value system is Bigtable [4], which continues to see massive usage at Google for a variety of applications. However, developers of many OLTP applications found it difficult to build these applications without a strong schema system, cross-row transactions, consistent replication and a powerful query language. The initial response to these difficulties was to build transaction processing systems on top of Bigtable; an example is Megastore [2]. While these systems provided some of the benefits of a database system, they lacked many traditional database features that application developers often rely on. A key example is a robust query language, meaning that developers had to write complex code to process and aggregate the data in their applications. As a result, we decided to turn Spanner into a full featured SQL system, with query execution tightly integrated with the other architectural features of Spanner (such as strong consistency and global replication). Spanner's SQL interface borrows ideas from the F1 [9] system, which was built to manage Google's AdWords data, and included a federated query processor that could access Spanner and other data sources.

Today, Spanner is widely used as an OLTP database management system for structured data at Google, and is publicly available in beta as Cloud Spanner[1] on the Google Cloud Platform (GCP). Currently, over 5,000 databases run in our production instances, and are used by teams across many parts of Google and its parent company Alphabet. This data is the "source of truth" for a variety of mission-critical Google databases, incl. AdWords. One of our large users is the Google Play platform, which executes SQL queries to manage customer purchases and accounts. Spanner serves tens of millions of QPS across all of its databases, managing hundreds of petabytes of data. Replicas of the data are served from datacenters around the world to provide low latency to scattered clients. Despite this wide replication, the system provides transactional consistency and strongly consistent replicas, as well as high availability. The database features of Spanner, operating at this massive scale, make it an attractive platform for new development as well as migration of applications from existing data stores, especially for "big" customers with lots of data and large workloads. Even "small" customers benefit from the robust database features, strong

SIGMOD'17, May 14–19, 2017, Chicago, IL, USA

ACM ISBN 978-1-4503-4197-4/17/05.

DOI: http://dx.doi.org/10.1145/3035918.3056103

[1] http://cloud.google.com/spanner

mongoDB

MONGODB

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

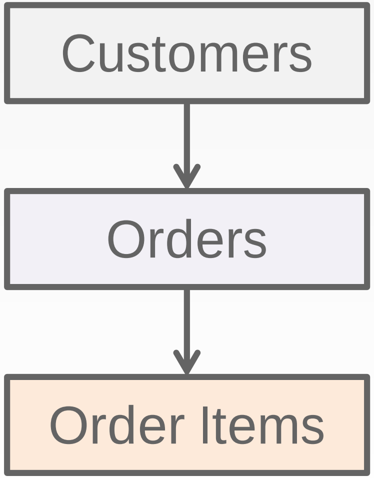
Open-source (Server Side Public License)

Centralized shared-nothing architecture.

Concurrency Control:
→ OCC with multi-granular locking

PHYSICAL DENORMALIZATION

A customer has orders and each order has order items.



$R_1(\texttt{custId},\texttt{name},\ldots)$

$\bowtie$

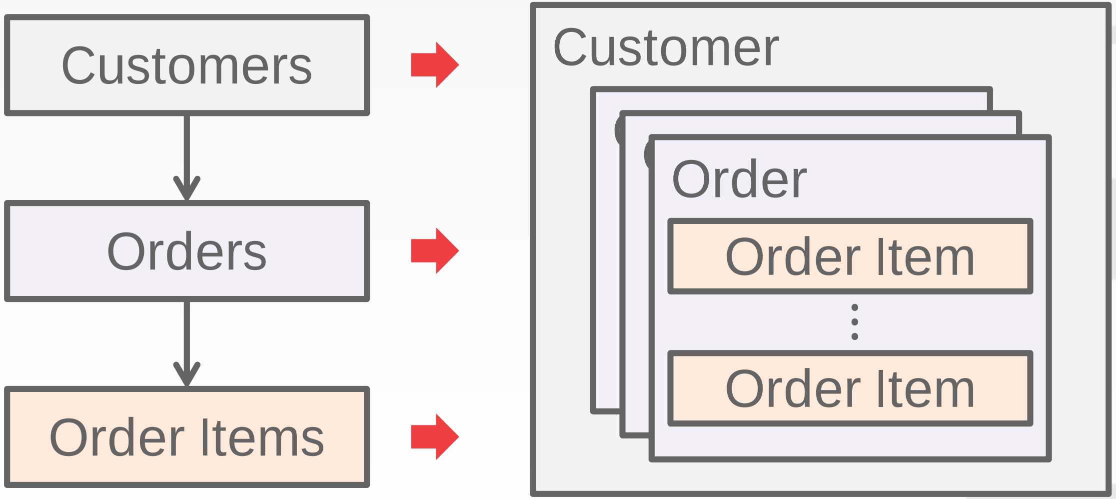
$R_2(\texttt{orderId},\texttt{custId},\ldots)$

$\bowtie$

$R_3(\texttt{itemId},\texttt{orderId},\ldots)$

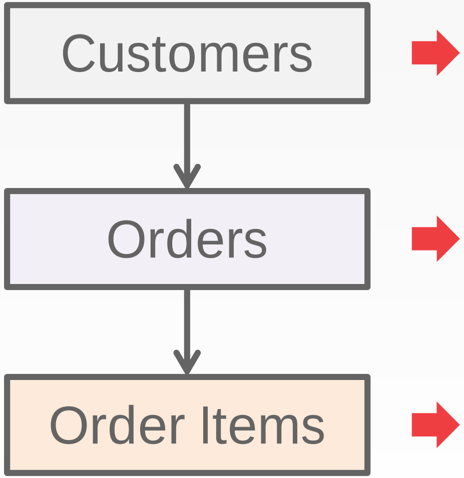
PHYSICAL DENORMALIZATION

A customer has orders and each order has order items.



PHYSICAL DENORMALIZATION

A customer has orders and each order has order items.




```
{
  "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 (new in 2018).

DISTRIBUTED ARCHITECTURE

Heterogeneous distributed components.

→ Shared nothing architecture

→ Centralized query router.

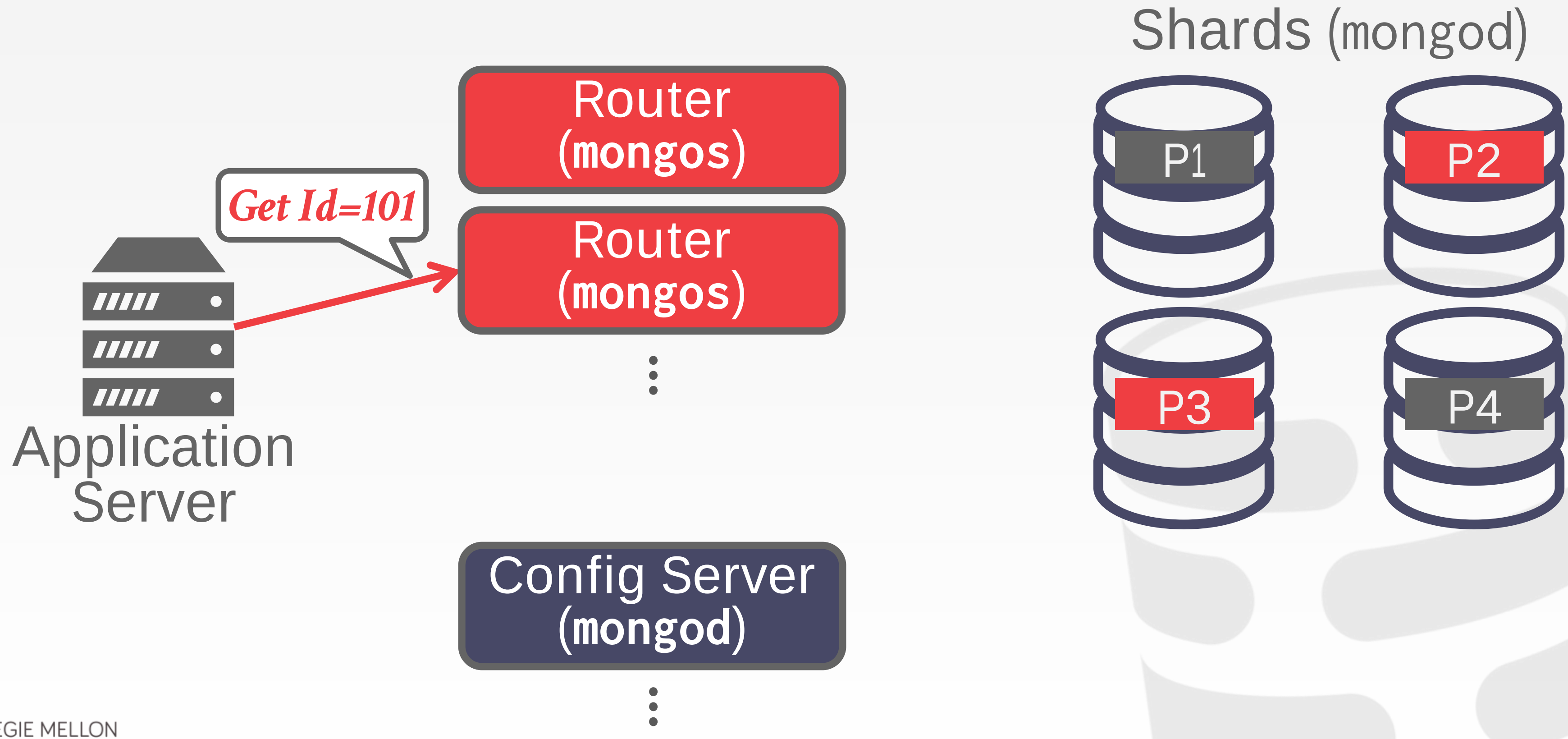
Master-slave replication.

Auto-sharding:

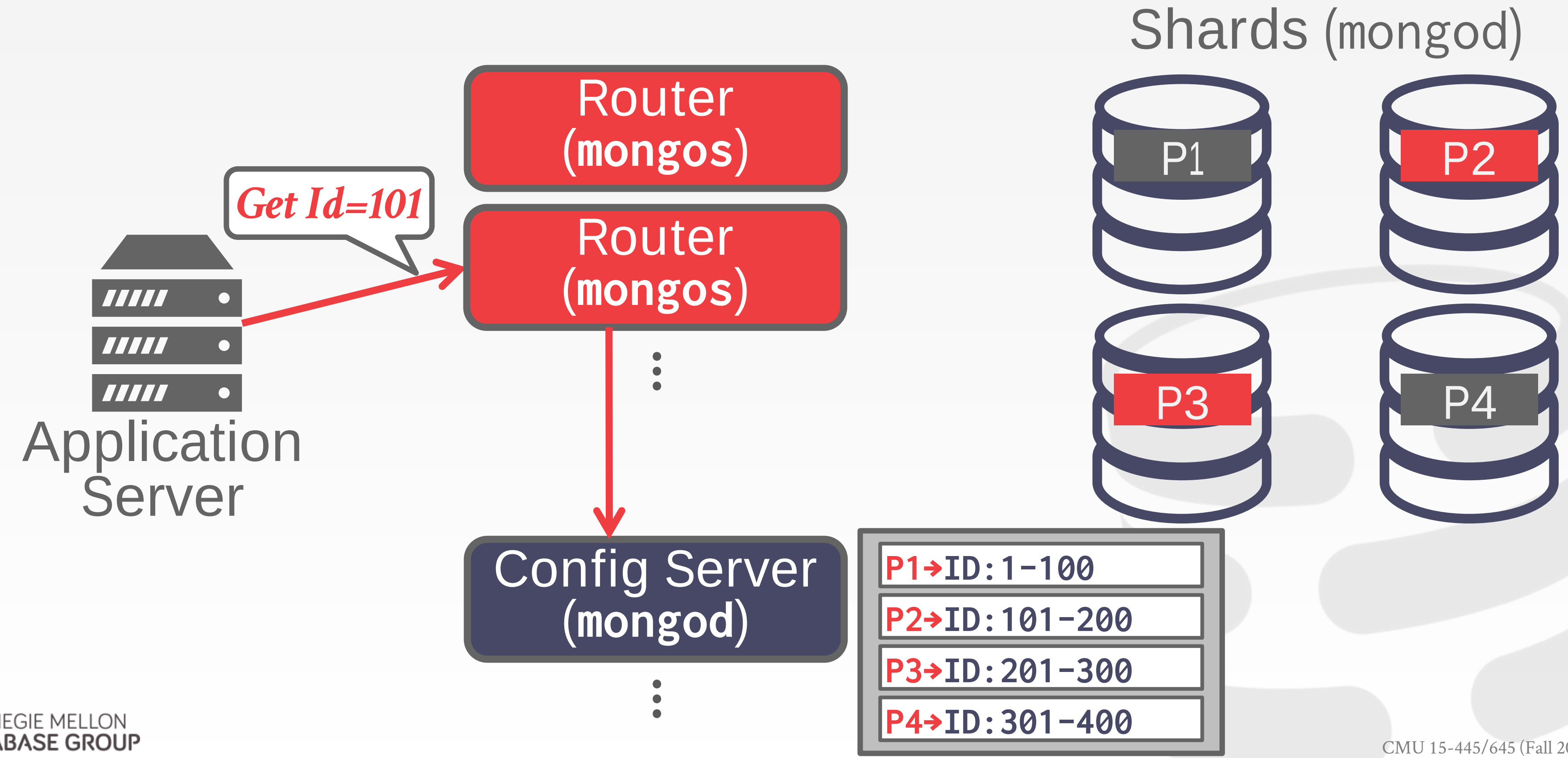
→ Define 'partitioning' attributes for each collection (hash or range).

→ When a shard gets too big, the DBMS automatically splits the shard and rebalances.

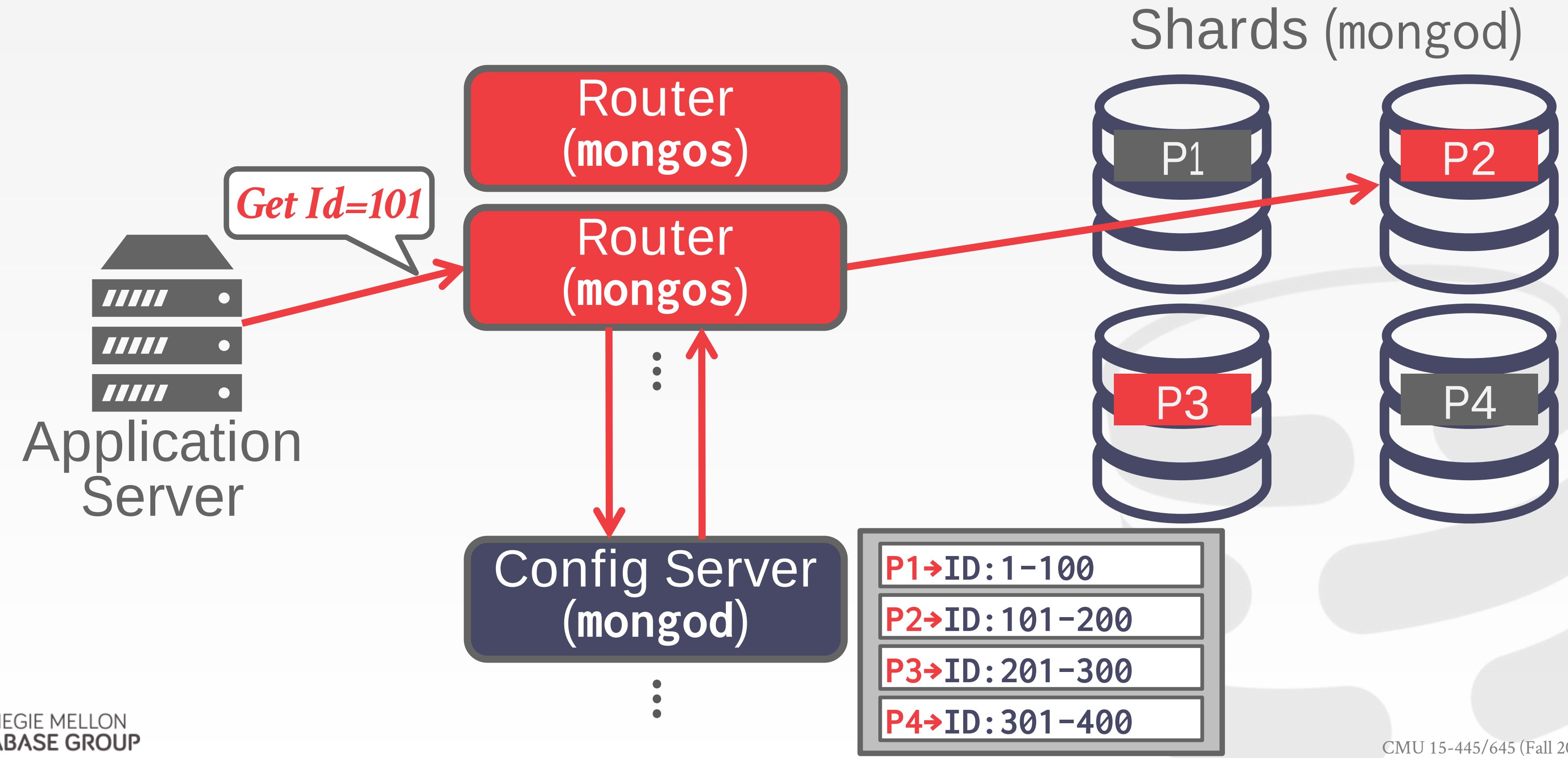
MONGODB CLUSTER ARCHITECTURE



MONGODB CLUSTER ARCHITECTURE



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

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.