Advanced Data Management (CSCI 490/680)

Scalable Databases

Dr. David Koop
3.2 Shared-Nothing

A shared-nothing parallel system (Figure 3.2) is made up of a cluster of independent machines that communicate over a high-speed network interconnect or, increasingly frequently, over commodity networking components. There is no way for a given system to directly access the memory or disk of another system.

Shared-nothing systems provide no hardware sharing abstractions, leaving coordination of the various machines entirely in the hands of the DBMS. The most common technique employed by DBMSs to support these clusters is to run their standard process model on each machine, or node, in the cluster. Each node is capable of accepting client SQL requests in parallel. The main challenge is to modify the query execution layers to take advantage of the ability to parallelize a single query across multiple CPUs; we defer this to Section 5.

Parallel DB Architecture: Shared Nothing

[Figure 3.2 Shared-nothing architecture.]

[Hellerstein et al., Architecture of a Database System]
Sharding

Collection 1

Shard A
Collection 1

Shard B

Shard C

Shard D

1 TB

256 GB

256 GB

256 GB

256 GB
Stonebraker: The End of an Architectural Era

- "RDBMSs were designed for the business data processing market, which is their sweet spot"
- "They can be beaten handily in most any other market of significant enough size to warrant the investment in a specialized engine"
- Changes in markets (science), necessary features (scalability), and technology (amount of memory)
- RDBMS Overhead: Logging, Latching, and Locking
- Relational model is not necessarily the answer
- SQL is not necessarily the answer
Problems with Relational Databases

ID: 1001
Customer: Ann

Line Items:
- 0321293533 2 $48 $96
- 0321601912 1 $39 $39
- 0131495054 1 $51 $51

Payment Details:
Card: Amex
CC Number: 12345
Expiry: 04/2001

Orders
Customers
Order Lines
Credit Cards

[P. Sagalage]
## Horizontal Partitioning vs. Vertical Partitioning

### Vertical Partitions

<table>
<thead>
<tr>
<th>VP1</th>
<th>VP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER ID</td>
<td>FIRST NAME</td>
</tr>
<tr>
<td>1</td>
<td>TAEKO</td>
</tr>
<tr>
<td>2</td>
<td>O.V.</td>
</tr>
<tr>
<td>3</td>
<td>SELDA</td>
</tr>
<tr>
<td>4</td>
<td>JIM</td>
</tr>
</tbody>
</table>

### Horizontal Partitions

<table>
<thead>
<tr>
<th>HP1</th>
<th>HP2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER ID</td>
<td>FIRST NAME</td>
</tr>
<tr>
<td>1</td>
<td>TAEKO</td>
</tr>
<tr>
<td>2</td>
<td>O.V.</td>
</tr>
</tbody>
</table>

---

Original Table

<table>
<thead>
<tr>
<th>CUSTOMER ID</th>
<th>FIRST NAME</th>
<th>LAST NAME</th>
<th>FAVORITE COLOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TAEKO</td>
<td>OHNUKI</td>
<td>BLUE</td>
</tr>
<tr>
<td>2</td>
<td>O.V.</td>
<td>WRIGHT</td>
<td>GREEN</td>
</tr>
<tr>
<td>3</td>
<td>SELDA</td>
<td>BAĞCAN</td>
<td>PURPLE</td>
</tr>
<tr>
<td>4</td>
<td>JIM</td>
<td>PEPPER</td>
<td>AUBERGINE</td>
</tr>
</tbody>
</table>

---

M. Drake
CAP Theorem

Scalability: CAP Theorem

Remains accessible and operational at all times.

Available

Traditional relational databases: PostgreSQL, MySQL, etc.

Pick Two!

Consistency
Commits are atomic across the entire distributed system.

CP
HBase
MongoDB
Redis
MemcacheDB
BigTable-like systems

Partition Tolerance
Only a total network failure can cause the system to respond incorrectly.

P

Voidemort, Riak, Cassandra, CouchDB, Dynamo-like systems

AP

E. Brewer
Cassandra and CAP

Partition Tolerance

Availability

Consistency (ACID)

Atomicity
Consistency
Isolation
Durability

RDBMS

cassandra

G. Atil
What is Cassandra?

- Fast Distributed (Column Family NoSQL) Database
  - High availability
  - Linear Scalability
  - High Performance
- Fault tolerant on Commodity Hardware
- Multi-Data Center Support
- Easy to operate
- Proven: CERN, Netflix, eBay, GitHub, Instagram, Reddit
NoSQL: Column Stores

- Instead of having rows grouped/sharded, we group columns
- ...or families of columns
- Put similar columns together
- Examples: Cassandra, HBase
## Relational Databases vs. Cassandra

<table>
<thead>
<tr>
<th>Relational Database</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handles moderate incoming data velocity</td>
<td>Handles high incoming data velocity</td>
</tr>
<tr>
<td>Data arriving from one/few locations</td>
<td>Data arriving from many locations</td>
</tr>
<tr>
<td>Manages primarily structured data</td>
<td>Manages all types of data</td>
</tr>
<tr>
<td>Supports complex/nested transactions</td>
<td>Supports simple transactions</td>
</tr>
<tr>
<td>Single points of failure with failover</td>
<td>No single points of failure; constant uptime</td>
</tr>
<tr>
<td>Supports moderate data volumes</td>
<td>Supports very high data volumes</td>
</tr>
<tr>
<td>Centralized deployments</td>
<td>Decentralized deployments</td>
</tr>
<tr>
<td>Data written in mostly one location</td>
<td>Data written in many locations</td>
</tr>
<tr>
<td>Supports read scalability (with consistency sacrifices)</td>
<td>Supports read and write scalability</td>
</tr>
<tr>
<td>Deployed in vertical scale up fashion</td>
<td>Deployed in horizontal scale out fashion</td>
</tr>
</tbody>
</table>
Cassandra: Replication
Cassandra: Consistency Levels

- Data is always replicated according to replication factors
- Consistency Levels: ANY (only writes), ONE, LOCAL_ONE, QUORUM, LOCAL_QUORUM
- Consistency levels define how many replicas must fulfill the request
- LOCAL_* are local to the data center, others go across data centers
- quorum = (sum-of-replication-factors / 2) + 1
  - Each data center may have its own replication factor
- ANY provides lowest consistency but highest availability
- ALL provides the highest consistency and lowest availability (not recommended)
Multiple Data Center Replication

DC 1

DC 2

Write
Reading Response

• Spanner: Google's Globally-Distributed Database
Assignment 4

- Work on Data Integration and Data Fusion
- Integrate artist datasets from different institutions (The Met, The Tate, Smithsonian, Carnegie Museum of Art)
  - Integrate information about names, places, nationality, etc.
- Record Matching:
  - Which artists are the same?
  - Which nationalities are the same? (British/English)
- Data Fusion:
  - Year of birth/death differences
  - Nationality differences
Test 2

• Wednesday, April 6
• Covers material from the beginning of course, emphasizing material since Test 1
• Similar Format to Test 1
• We have discussed more papers since Test 1
NewSQL

A. Pavlo
Spanner:
Google's Globally-Distributed Database

J. C. Corbett et al.
Spanner Overview

- Focus on scaling databases focused on OLTP (not OLAP)
- Since OLTP, focus is on sharding **rows**
- Tries to satisfy CAP (which is impossible per CAP Theorem) by not worrying about 100% availability
- External consistency using multi-version concurrency control through timestamps
- ACID is important
- Structured: universe with zones with zone masters and then spans with span masters
- SQL-like (updates allow SQL to be used with Spanner)
Spanner and the CAP Theorem

Which type of system is Spanner?
- C: consistency, which implies a single value for shared data
- A: 100% availability, for both reads and updates
- P: tolerance to network partitions

Which two?
- CA: close, but not totally available
- So actually CP
Spanner Server Organization

[Corbett et al., 2012]
Interleaved Schema

CREATE TABLE Users {
    uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
    uid INT64 NOT NULL, aid INT64 NOT NULL, name STRING
} PRIMARY KEY (uid, aid), INTERLEAVE IN PARENT Users ON DELETE CASCADE;

<table>
<thead>
<tr>
<th>Users(1)</th>
<th>Directory 3665</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albums(1,1)</td>
<td></td>
</tr>
<tr>
<td>Albums(1,2)</td>
<td></td>
</tr>
<tr>
<td>Users(2)</td>
<td>Directory 453</td>
</tr>
<tr>
<td>Albums(2,1)</td>
<td></td>
</tr>
<tr>
<td>Albums(2,2)</td>
<td></td>
</tr>
<tr>
<td>Albums(2,3)</td>
<td></td>
</tr>
</tbody>
</table>

[Corbett et al., 2012]
External Consistency

- Traditional DB solution: **two-phase locking**—no writes while client reads
- "The system behaves as if all transactions were executed sequentially, even though Spanner actually runs them across multiple servers (and possibly in multiple datacenters) for higher performance and availability" [Google]
- Semantically indistinguishable from a single-machine database
- Uses multi-version concurrency control (MVCC) using **timestamps**
- Spanner uses **TrueTime** to generate monotonically increasing timestamps across all nodes of the system
TrueTime

- API to try to keep computers on a globally-consistent clock
- Uses GPS and Atomic Clocks!
- Time masters per datacenter (usually with GPS)
- Each machine runs a timeslave daemon
- Armageddon masters have atomic clocks
- API:

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT.now()</td>
<td>TTinterval: [earliest, latest]</td>
</tr>
<tr>
<td>TT.after(t)</td>
<td>true if t has definitely passed</td>
</tr>
<tr>
<td>TT.before(t)</td>
<td>true if t has definitely not arrived</td>
</tr>
</tbody>
</table>

[Corbett et al., 2012]
Concurrent Control

- Use TrueTime to implement concurrency control
- Types of reads and writes:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Timestamp Discussion</th>
<th>Concurrency Control</th>
<th>Replica Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Write Transaction</td>
<td>§ 4.1.2</td>
<td>pessimistic</td>
<td>leader</td>
</tr>
<tr>
<td>Read-Only Transaction</td>
<td>§ 4.1.4</td>
<td>lock-free</td>
<td>leader for timestamp; any for read, subject to § 4.1.3</td>
</tr>
<tr>
<td>Snapshot Read, client-provided timestamp</td>
<td>—</td>
<td>lock-free</td>
<td>any, subject to § 4.1.3</td>
</tr>
<tr>
<td>Snapshot Read, client-provided bound</td>
<td>§ 4.1.3</td>
<td>lock-free</td>
<td>any, subject to § 4.1.3</td>
</tr>
</tbody>
</table>

- Use Two-Phase Commits (2PC)

[Corbett et al., 2012]
Two-Phase Commit Scalability

<table>
<thead>
<tr>
<th>participants</th>
<th>latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
</tr>
<tr>
<td>1</td>
<td>17.0 ±1.4</td>
</tr>
<tr>
<td>2</td>
<td>24.5 ±2.5</td>
</tr>
<tr>
<td>5</td>
<td>31.5 ±6.2</td>
</tr>
<tr>
<td>10</td>
<td>30.0 ±3.7</td>
</tr>
<tr>
<td>25</td>
<td>35.5 ±5.6</td>
</tr>
<tr>
<td>50</td>
<td>42.7 ±4.1</td>
</tr>
<tr>
<td>100</td>
<td>71.4 ±7.6</td>
</tr>
<tr>
<td>200</td>
<td>150.5 ±11.0</td>
</tr>
</tbody>
</table>

[Corbett et al., 2012]
Distribution of TrueTime Epsilons

The data shows that these two factors in determining datacenter for routine maintenance. We continue to in-
creasing on March 30 were due to networking improvements
higher values of
ever, there can be significant tail-latency issues that cause
the base value of
measures time-master uncertainty (which is generally 0)
tooth in
each of
90
99
99.9

Figure 6:

Table 5 illustrates the distribution of the number of
# fragments
100–500
5336
15–99
341
10–14
34
2–4
7
1

dates in external Bigtables, which compromised transac-
practical. Application semantics requires transactions
semantics, which made using other NoSQL systems im-
and downtime. Third, F1 requires strong transactional
replication, failover was difficult, and risked data loss
failure in the last few months, the most that the F1 team
ble to them. Although there have been unplanned cluster
 correlates, and also the choice of their frontend sites. Anecdo-
to cope with outages due to potential major natural disas-
3 on the east coast. This choice of replica sites was made
stack. F1 has 2 replicas on the west coast of the US, and
F1 to Spanner, instead of the MySQL-based application
indexes), and was able to implement their own consistent
indexes on their data (since Span-
also needed secondary indexes on their data (since Span-
across arbitrary data, and consistent reads. The F1 team
(required by applications) that use the F1 global indexes using Spanner transactions.
F1 has some knowledge of the sharding in application
query processing on a per-customer basis, but
fixed shard. This layout enabled the use of indexes and
scheme assigned each customer and all related data to a
difficulties with sharded MySQL. The MySQL sharding
many NoSQL instances, but was large enough to cause
dataset is tens of terabytes, which is small compared to
was manually sharded many ways. The uncompressed
backend was originally based on a MySQL database that
does not yet provide automatic support for secondary
Table 5:

<table>
<thead>
<tr>
<th>Epsilon (ms)</th>
<th>Mar 29</th>
<th>Mar 30</th>
<th>Mar 31</th>
<th>Apr 1 6AM</th>
<th>8AM</th>
<th>10AM</th>
<th>12PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Corbett et al., 2012]
F1: A Distributed SQL Database That Scales

F1: OLTP and OLAP Together

- Distributed data storage: data is not stored at one central location
- Need to keep data and schemas in sync
- Hierarchical schemas keep data that is likely to be accessed at the same time together
- Optimistic Transactions: Long reads that keep track of timestamps and don't lock the database until the write happens
- Change History: Keep track of history with database, also helps with caching
- DIY Object-Relational Mapping: don't automatically join or implicitly traverse relationships
- Protocol buffers as a way to store application data without translation + support for queries
Hierarchical Schema

Explicit table hierarchies. Example:

- **Customer** (root table): PK (CustomerId)
- **Campaign** (child): PK (CustomerId, CampaignId)
- **AdGroup** (child): PK (CustomerId, CampaignId, AdGroupId)

### Rows and PKs

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>1,3</td>
<td>2,5</td>
</tr>
<tr>
<td>Campaign</td>
<td>1,4</td>
<td></td>
</tr>
<tr>
<td>AdGroup</td>
<td>1,3,5</td>
<td>2,5,8</td>
</tr>
<tr>
<td></td>
<td>1,3,6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,4,7</td>
<td></td>
</tr>
</tbody>
</table>

### Storage Layout

- **Customer** (1)
- **Campaign** (1,3)
- **AdGroup** (1,3,5)
- **AdGroup** (1,3,6)
- **Campaign** (1,4)
- **AdGroup** (1,4,7)
- **Customer** (2)
- **Campaign** (2,5)
- **AdGroup** (2,5,8)

[Shute et al., 2012]
Clustered Storage

- Child rows under one root row form a **cluster**
- Cluster stored on one machine (unless huge)
- Transactions within one cluster are most efficient
- Very efficient joins inside clusters (can merge with no sorting)

---

**Rows and PKs**

```
1
| 1,3 | 1,4 |
1,3,5 | 1,3,6 | 1,4,7 |
```

---

**Storage Layout**

```
Customer (1)
Campaign (1,3)
AdGroup (1,3,5)
AdGroup (1,3,6)
Campaign (1,4)
AdGroup (1,4,7)
Customer (2)
Campaign (2,5)
AdGroup (2,5,8)
```

[Shute et al., 2012]
F1 Notes

- Schema changes: allow two different schemas
- Transaction types: Snapshot, Pessimistic, Optimistic
- Change History and application to caching
- Disk latency or network latency?
Discussion
Google Cloud Spanner

- [https://cloud.google.com/spanner/](https://cloud.google.com/spanner/)

- **Features:**
  - Global Scale: thousands of nodes across regions / data centers
  - Fully Managed: replication and maintenance are automatic
  - Transactional Consistency: global transaction consistency
  - Relational Support: Schemas, ACID Transactions, SQL Queries
  - Security
  - Highly Available
Google Cloud Spanner: NewSQL

<table>
<thead>
<tr>
<th></th>
<th>Cloud Spanner</th>
<th>Traditional Relational</th>
<th>Traditional Non-Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SQL</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Consistency</td>
<td>Strong</td>
<td>Strong</td>
<td>Eventual</td>
</tr>
<tr>
<td>Availability</td>
<td>High</td>
<td>Failover</td>
<td>High</td>
</tr>
<tr>
<td>Scalability</td>
<td>Horizontal</td>
<td>Vertical</td>
<td>Horizontal</td>
</tr>
<tr>
<td>Replication</td>
<td>Automatic</td>
<td>Configurable</td>
<td>Configurable</td>
</tr>
</tbody>
</table>

[https://cloud.google.com/spanner/]
write the code to handle outage exceptions: if they haven’t written that code, then they are assuming high availability. Based on a large number of internal users of Spanner, we know that they assume Spanner is highly available.

In addition to Spanner (“fate sharing”). We actually care about the differential availability, in which the user is up (and making a request) to notice that Spanner is down. This number is strictly higher (more available) than Spanner’s actual availability — that is, you have to hear the tree fall to count it as a problem.

A third issue is whether or not outages are due to partitions. If the primary causes of Spanner outages are not partitions, then CA is in some sense more accurate. For example, any database cannot provide availability if all of its replicas are offline, which has nothing to do with partitions. Such a multi-replica partitions as a factor in availability. For Spanner, this means that when there is an availability outage, it is not in practice due to a partition, but rather some other set of multiple faults (as no single fault will forfeit availability).

Availability data
Before we get to Spanner, it is worth taking a look at the evolution of Chubby, another wide-area system that provides both consistency and availability. The original Chubby paper [Bur06] mentioned nine outages of 30 seconds or more in 700 days, and six of those were network related (as discussed in [BK14]). This corresponds to an availability worse than 5 9s (at best), to a more realistic 4 9s if we assume an average of 10 minutes per outage, and potentially even 3 9s at hours per outage.

For locking and consistent read/write operations, modern geographically distributed Chubby cells provide an average availability of 99.99958% (for 30s+ outages) due to various network, architectural and operational improvements. Starting in 2009, due to “excess” availability, Chubby’s Site Reliability Engineers (SREs) started forcing periodic outages to ensure we continue to understand dependencies and the impact of Chubby failures.

Internally, Spanner provides a similar level of reliability to Chubby; that is, better than 5 9s. The Cloud version has the same foundation, but adds some new pieces, so it may be a little lower in practice for a while.

Causes of Spanner Availability Incidents

- 52.5% User
- 7.6% Network
- 10.9% Other
- 12.1% Cluster
- 13.3% Bug
- 3.7% Operator

[Causes of Spanner Availability Incidents chart]

[E. Brewer, 2017]
Causes of Spanner Incidents

- User: overload or misconfiguration (specific to one user)
- Cluster: non-network problems, e.g. servers and power
- Operator: misconfiguration by people
- Bug: software error that caused some problem
- Other: most are one-offs
- Network: individual data centers/regions cut off and under-provisioned bandwidth, uni-directional traffic

[E. Brewer, 2017]
Spanner as "Effectively CA"

- Criteria for being "effectively CA"
  1. At a minimum it must have very high availability in practice (so that users can ignore exceptions), and
  2. As this is about partitions it should also have a low fraction of those outages due to partitions.
- Spanner meets both of these criteria
- Spanner relies on Google's network (private links between data centers)
- TrueTime helps create consistent snapshots, sometimes have a commit wait

[E. Brewer, 2017]
## More Recent Tests: Spanner vs. MySQL

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.30%</td>
</tr>
<tr>
<td>2</td>
<td>0.25%</td>
</tr>
<tr>
<td>3</td>
<td>4.22%</td>
</tr>
<tr>
<td>4</td>
<td>1.88%</td>
</tr>
<tr>
<td>5</td>
<td>3.28%</td>
</tr>
<tr>
<td>6</td>
<td>14.13%</td>
</tr>
<tr>
<td>7</td>
<td>12.56%</td>
</tr>
<tr>
<td>8</td>
<td>0.49%</td>
</tr>
<tr>
<td>9</td>
<td>4.11%</td>
</tr>
<tr>
<td>10</td>
<td>0.43%</td>
</tr>
<tr>
<td>11</td>
<td>0.59%</td>
</tr>
<tr>
<td>12</td>
<td>36.76%</td>
</tr>
<tr>
<td>13</td>
<td>0.61%</td>
</tr>
<tr>
<td>14</td>
<td>6.10%</td>
</tr>
<tr>
<td>15</td>
<td>0.33%</td>
</tr>
<tr>
<td>16</td>
<td>12.56%</td>
</tr>
<tr>
<td>17</td>
<td>1.06%</td>
</tr>
<tr>
<td>18</td>
<td>0.32%</td>
</tr>
</tbody>
</table>

[P. Bakkum and D. Cepeda, 2017]
Latency: Spanner vs. MySQL

Latency at 3,000 Queries per Second

[P. Bakkum and D. Cepeda, 2017]
Latency: Spanner vs. MySQL

Latency at 9,000 Queries per Second

Median Latency (ms)

Query

[Latency: Spanner vs. MySQL chart showing comparison between Spanner and MySQL across different queries, with median latency in milliseconds.]

[P. Bakkum and D. Cepeda, 2017]
Throughput: Spanner vs. MySQL

Median Latency as Throughput Increases

- MySQL (median)
- spanner 9 nodes (median)
- spanner 15 nodes (median)
- spanner 30 nodes (median)

[P. Bakkum and D. Cepeda, 2017]
Max Throughput vs. Nodes

[P. Bakkum and D. Cepeda, 2017]
Spanner: Latency vs. Nodes

Latency at 3000 QPS vs Nodes

Latencies @ 3000 QPS

Nodes

[Note: The graph shows the latency in seconds at 3000 QPS (queries per second) for different numbers of nodes. The latency decreases as the number of nodes increases, indicating improved performance.]